von Karman Institute for Fluid Dynamics

Lecture Series 1997-02

28th computational fluid dynamics

3-7 March, 1997

Solution Adaptive Cartesian Grid Methods for Aerodynamic Flows with Complex Geometries

M. J. Aftosmis

USAF/NASA Ames Research Center
Mail Stop T27B-2
Moffett Field, CA 94035-1000

Lecture notes for 28th Computational Fluid Dynamics Lecture Series, von Karman Institute for Fluid Dynamics, Chaussée de Waterloo 72, B-1640 Rhode-Saint-Genèse, Belgium

Appendix B

Contents

Solution adaptive Cartesian grid methods for aerodynamic flows with complex geometries

Michael J. Aftosmis

Wright-Laboratory / NASA Ames
Mail Stop T-27B-2
NASA Ames Research Center
Moffett Field, CA 94035-1000

Cartesian methods for CFD offer an accurate and robust approach for simulating aerodynamic flows around geometrically complex bodies. As a result of this flexibility, the quantity of literature devoted to their study has grown substantially in recent years. These notes attempt to cover only a subset of this on-going research. In doing so, however, they aim to provide insight into the fundamental challenges faced by practitioners of the approach, while also serving as a guide for further exploration. The integration schemes used in Cartesian solvers are similar to those used in other approaches. Therefore, these notes focus mainly on the geometric algorithms, surface modeling and boundary conditions needed to design a successful Cartesian mesh scheme.

1. Overview

1.1 Introduction

While both structured and unstructured approaches for CFD have enjoyed reasonable success in their application to real-world problems, neither method has offered a truly "automatic" method for discretizing the domain around arbitrarily complex geometries. One reason for this stems from the fact that both techniques are *body-fitted* - *i.e.* cells neighboring the body must conform to the surface. This implies that the connectivity of the computational mesh is intimately linked to the body's geometry and topology. As a result, the surface mesh is (sometimes) subject to conflicting requirements of resolving both the local geometry and the expected flow variation. While unstructured surface triangulations relieve some of this burden by permitting vertices of arbitrary degree, structured surface meshes have the additional constraint of a prescribed connectivity.

Cartesian methods differ in that they are *non*-body-fitted. Hexahedral cells in the domain consist of a set of right parallelepipeds and the (normally) orthogonal grid system may extend through solid wall boundaries within the computational domain. The method then removes, or flags, cells which are completely internal to the geometry and identifies those

cells which intersect the solid walls. The remaining cells are considered general volume mesh elements. Fundamentally, Cartesian approaches trade the case-specific problem of generating a body-fitted surface mesh for the more general problem of computing and characterizing intersections between hexahedral cells and the surface geometry. Thus, all difficulties associated with meshing a particular geometry are restricted to a lower order manifold which constitutes the wetted surface of the geometry. Researchers have been very successful in applying these techniques to extremely complex geometries and have demonstrated that the technique is very amenable to automation[34,47,56,57,58,74,91].

The generality of this approach has an important implication when assessing the surface modeling requirements of a Cartesian method. Since mesh cells cut the geometry arbitrarily, the cut-cells at the surface are *de-coupled* from the description of the surface itself. The surface description is no longer required to resolve both the flow and the local geometry as is the case with body-fitted approaches. The surface description therefore may focus uniquely on the task of resolving the geometry, while the mesh cells handle the job of describing the flow. We note at the outset, however, that since solid wall boundaries do cut arbitrarily through the layer of “cut-cells” encasing the body, accurate surface boundary conditions play an obvious role in successful Cartesian schemes.

Cartesian approaches fall into two general categories. Either they consider the mesh as an unstructured (or octree structured) collection of *h*-refined hexahedra, or they operate by embedding structured sub-grids within structured mesh blocks (see the Adaptive Mesh Refinement (AMR) work of [34]or [74]). In the unstructured or octree techniques, volume meshing of the computational domain relies on the simple and robust procedure of cell division. Beginning with a coarse background grid - or even a single root cell - hexahedral elements are repeatedly subdivided in order to resolve emerging features of the flow or geometry. The final mesh is viewed as either completely unstructured, or with underlying octree connectivity. Structured approaches embed *i,j,k* structured patches with similar resolution goals. Successful 2-D and 3-D solution procedures have been proposed following both implementations[18,34,56,74]

1.2 Development of Cartesian Mesh Approaches[1]

While three dimensional applications to complex geometry have become commonplace only recently, Cartesian approaches have been evaluated since the late 1970’s.

1. This discussion follows that of Melton in [56].

Work by Purvis and Burkhalter[73] solved the full potential equation on 2D Cartesian meshes using a finite volume method. Solution of the Euler equations was pursued in the mid-1980's by a many researchers[29,45] and the first three dimensional inviscid solutions appeared in the late 1980's by Gaffney, Hassan and Salas[42].

Cartesian approaches have been successfully utilized in industrial applications including Boeing's TRANAIR code which solves the full potential equation, and the commercially available MGAERO[85] package for Euler simulations. These applications are notable because they provide close links with surface modeling and provide a wide base of experience with large-scale computations using Cartesian methods. MGAERO, for example, adopts a component-based approach to complex geometries[51,52]. Similar approaches have been pursued by various other groups[58,3] and will be discussed in §1.3.3.

The cut-cells which are necessarily present in Cartesian discretizations present unique problems in the implementation of accurate boundary conditions. While these notes will discuss this topic in some detail, references [18], [17], [16], [32], [40], [41], and [56] (among others) present additional insight into the issues involved.

The isotropic elements stemming from *h*-refinement of Cartesian hexahedra are well suited to resolving flow structures in inviscid simulations. However, use of such elements to capture boundary layers and other stiff viscous phenomena would be grossly inefficient[43,31]. Recently, a variety of authors have proposed alternate techniques for extending inviscid Cartesian approaches to viscous flow[47,31]. Historically, the lack of a clear extension to viscous simulations has been a weak link and the recently proposed avenues offer the possibility of further research.

1.3 Motivation for Cartesian Mesh Approaches

Arguments in favor of Cartesian approaches generally rely on claims of either increased accuracy or their flexibility when applied to complex geometries. In this section we present examples of these arguments while also discussing shortcomings of the approach.

1.3.1 Accuracy of Difference Schemes on Uniform and non-Uniform Meshes

Away from cut-cell boundaries and mesh refinement interfaces, Cartesian methods integrate the governing equations using a uniform, orthogonal discretization of space. This guarantee of mesh quality suggests improved accuracy for a given difference scheme. Improved accuracy translates into lower discretization error for a given

number of cells. This can also be interpreted as an efficiency claim, since it implies that a smaller number of cells will be required to achieve a specified level of numerical accuracy.

A simple model problem demonstrates the basis for this claim. The Euler and Navier-Stokes equations are modeled with the advection equation, and we examine its behavior in a single dimension. Such simplifications are warranted since this demonstration proceeds by induction. Rather than show that Cartesian approaches have a particular advantage for scalar 1-D systems (which may or may not break down in their extension to multiple dimensions and coupled systems) we intend to show that even on the simplest 1-D mesh systems, ordinary difference schemes show weaknesses on non-uniform meshes, and that these difficulties persist for systems and in multi-dimensions[1].

Consider the scalar advection equation with unit wave speed in a single dimension over the interval from $0 \le x \le 1$.

$$u_t + u_x = 0 \qquad 0 \le x \le 1 \tag{1.1}$$

The simplest consistent discretization for this equation is Godunov's method which uses first order spatial derivatives and forward Euler time advancement. On a uniform mesh with a spacing of h and time step Δt, this scheme may be written:

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{h}(u_i^n - u_{i-1}^n) \tag{1.2}$$

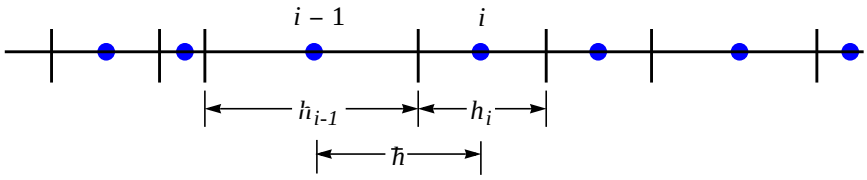


Figure 1-1: Nomenclature on a non-uniform mesh in 1-D.

Figure 1-1 shows an irregular mesh for discussion. Generalizing the difference scheme in eq. 1.2 to retain conservation gives:

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{h_i}(u_i^n - u_{i-1}^n) = Qu_i^n \tag{1.3}$$

1. The analysis in this section follows a similar presentation in Reference [13].

We examine the scheme's accuracy on non-uniform meshes by substituting in the exact solution $u(x,t)$. The local truncation error (LTE) of the scheme at time t^{n+1} in the vicinity of x_i is the difference of the exact solution and the discrete representation in eq. 1.3; $u(x_i, t^{n+1}) - Qu(x_i, t^{n+1}) = \Delta t \cdot LTE$.

$$\Delta t \cdot LTE = u(x_i, t^{n+1}) - u(x_i, t^n) + \frac{\Delta t}{h_i}\left[u(x_i, t^n) - u(x_{i-1}, t^n)\right] \tag{1.4}$$

Taylor expanding for data in the neighborhood of x_i gives:

$$\begin{aligned}\Delta t \cdot LTE = {} & u(x_i, t^n) + \Delta t u_t(x_i, t^n) + \frac{\Delta t^2}{2} u_{tt}(x_i, t^n) + \ldots - u(x_i, t^n) \\ & + \frac{\Delta t}{h_i}\left[u(x_i, t^n) - \left(u(x_i, t^n) - \hbar u_x(x_i, t^n) + \frac{\hbar^2}{2} u_{xx}(x_i, t^n) + \ldots\right)\right]\end{aligned} \tag{1.5}$$

where $\hbar$ is the average mesh interval between i and $i+1$, $\hbar \equiv (h_i + h_{i-1})/2$. Dropping the higher order terms and converting spatial derivatives with eq. 1.1 gives an expression for the leading terms in the truncation error expression.

$$LTE \cong \left(\frac{\hbar}{h} - 1\right) u_x(x_i, t^n) + \left(\frac{\Delta t}{2} - \frac{\hbar^2}{2h_i}\right) u_{xx}(x_i, t^n) \tag{1.6}$$

If the mesh is smooth (*i.e.* $h_{i-1} = h_i - O(h_i^2)$) then eq. 1.6 becomes:

$$LTE \approx O(h_i) + O(\Delta t + h) \tag{1.7}$$

which implies that the method remains consistent on smooth meshes. Note, however, that the first term on the right side of eq. 1.6 vanishes completely for uniform meshes. Thus on non-uniform grids, the scheme will have greater local truncation error than it would on an equivalent uniform mesh[1].

Table 1.1 chronicles the results of a numerical experiment with the difference scheme of eq. 1.3 using a Gaussian pulse initial condition integrated until time $t = 0.8$ while maintaining $\Delta t/h = 0.8$. This experiment was conducted using a uniform mesh with N intervals and two meshes with intervals which were randomly perturbed by $0.1h$ and $0.25h$. Error is tabulated in measures of the L_1 norm as com-

1. On non-smooth meshes the leading term is $O(1)$ which implies that the method may become inconsistent when integrating to a predetermined time. However, reference [92] has demonstrated that the method does manage to remain consistent, albeit with substantially higher truncation error than on smooth or uniform meshes.

pared to the exact solution. This table shows that the presence of the additional error term on the right of eq. 1.6 increases the local truncation error by nearly 50% over that of the uniform mesh for a 10% perturbation, and that a perturbation of 25% essentially doubles the magnitude of the truncation error. Moreover, examination of the column labeled "Uniform 1.25*h*" reveals that a uniform mesh with a 25% larger average spacing consistently outperforms a non-uniform grid with only a 10% variation in cell size[1].

When one admits discontinuous solutions, the question of accuracy becomes more difficult to analyze directly. Nevertheless, results from references [63] and [93] indicate that traveling discontinuities have smaller phase errors on regular meshes where the shocks can relax into a discrete traveling wave.

Table 1.1. Accuracy of first-order scheme on uniform and non-uniform meshes[1]

# of cells *N*	**% Error in $\|L_1\|$**			
	Uniform (h)	Uniform (1.25h)	10% Perturbation	25% Perturbation
20	13%	16%	18%	24%
40	7.2%	8.8%	9.7%	13%
80	3.7%	4.6%	5.1%	7.4%
160	1.9%	2.4%	2.7%	3.9%

1.3.2 Complex Geometry

The classification of cells as either "body-intersecting" or "flow field" has a unifying effect within a Cartesian approach. It implies that the detail of an intersection is not important to the mesh generation process and that topological information does not play a role in the mesh generation scheme. Similarly, the method is not linked to a specific representation of boundary surfaces. 3-D geometry is frequently specified through CAD or Stereo Lithography (STL) data, trimmed NURBS elements, or component triangulations. Assuming the existence of an appropriate routine for clipping mesh cells against these datatypes, other details of the geometry specification remain inconsequential.

Figure 1-2 demonstrates the degree of geometric complexity that can be routinely considered by Cartesian mesh approaches. The example shows a Cartesian volume mesh with 5.81M cells discretizing the space around an attack helicopter configuration which includes armaments, powerplants, wing stores, night-vision equipment,

1. Data for this example were contributed by M.Berger, who presented similar results in Ref. [13].

avionics packages and other components. The complete configuration contains 82 separate components which were extracted from the CAD description to build the model. This mesh was generated in about 5min and 20 seconds on a moderately powered engineering workstation (RISC R10000 CPU at 195Mhz). The only user inputs required for computing this mesh were (1) the bounding box of the domain, and (2) a target number of cells for the mesh generator. Figure 1-3 contains computed isobars on this configuration resulting from an inviscid flow analysis (computed on a coarser mesh with ~1.2M cells).

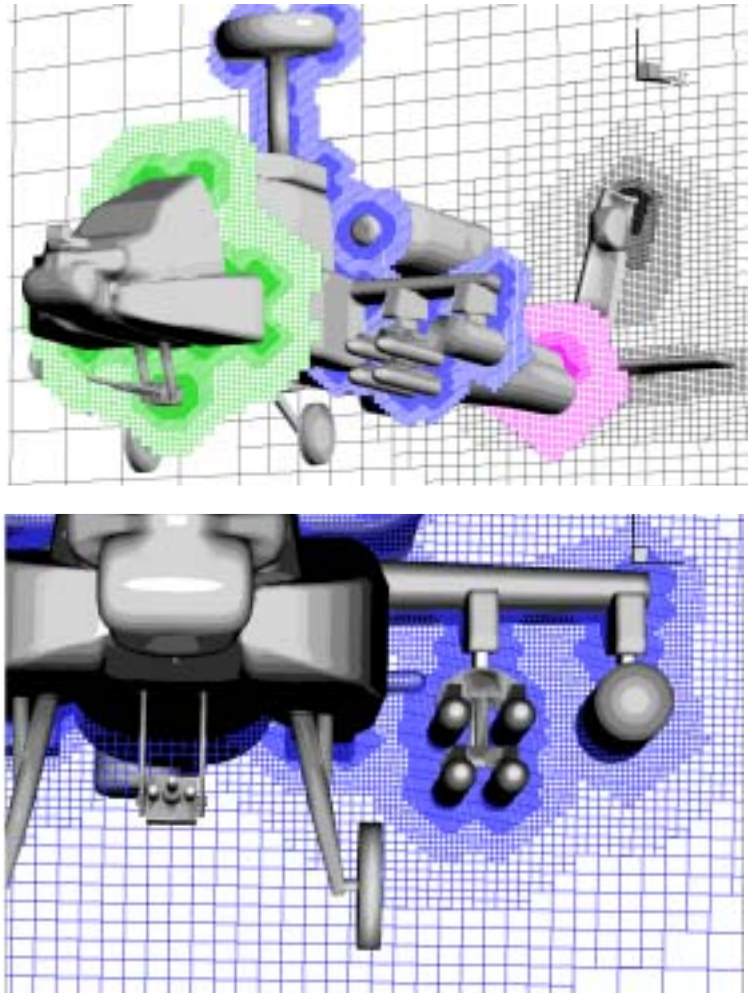


Figure 1-2: Upper: Cartesian mesh for attack helicopter configuration with 5.81M cells. Lower: Close-up of mesh through left wing and stores.

The example in Figures 1-2 and 1-3 was part of a study to optimize the shape of some of the packages on the wetted surface of this configuration. Such studies often require non-experts to run the mesh generation and flow solvers. The level of auto-

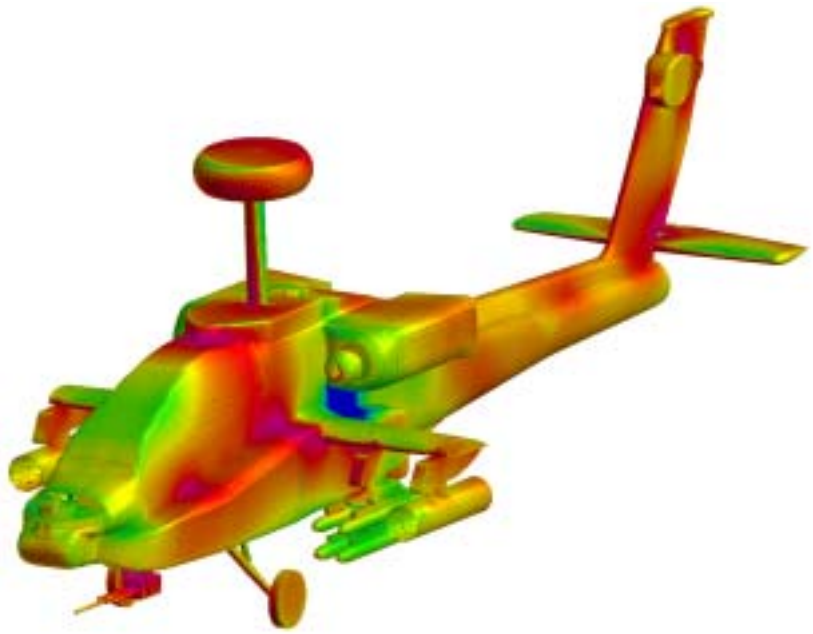


Figure 1-3: Isobars resulting from inviscid flow analysis of attack helicopter configuration computed on mesh with 1.2M cells

mation available with Cartesian approaches makes them particularly attractive for this type of analysis and optimization.

1.3.3 Surface Modeling Requirements

Given sufficient time and expertise, configurations like the helicopter in Figure 1-2 could also be analyzed with unstructured or even multi-block or overset structured solvers. In fact, unstructured methods may even produce tetrahedral meshes of similar or perhaps better quality than the Cartesian mesh shown. Such methods, however, generally have substantially higher surface modeling requirements, and with most automatic mesh generation schemes, surface modeling is typically the most (wall-clock) time consuming step in the process[1].

Surface modeling for unstructured (tetrahedral) methods plays two roles. First an input triangulation must be generated and constrained to the outer-moldline of the geometry. In addition, if two components are in close proximity to each other, their surface triangulations must have commensurate length scales and node distributions to permit a quality tetrahedralization of the space in between. Secondly, when mesh adaptation is applied, the volume or surface mesher must be able to insert new sites on the exterior surface of the boundary without creating invalid elements. These new sites must also conform to the actual surface. Generation of this constrained surface triangulation is non-trivial, especially when one considers that the

geometry may originally be specified via CAD data, loftings, natural/trimmed NURBS, grid patches, or a variety of other formats. Edges in this triangulation must follow wing leading and trailing edges, creases in the geometry, and especially intersections between components. To preserve such features, edge swaps must be restricted and constraint edges must be identified.

In unstructured mesh generation, creating such a constrained surface triangulation is typically the most time consuming and (user) interactive step in the process. Moreover, when a component is moved or reshaped, new intersections must be extracted, the surface mesh must be recomputed (at least locally) and a new volume mesh must be generated. If the surface triangulation is computed with a pre-processor or a commercial software package, then the real geometry will be unavailable for use by the adaptation routines, making boundary site insertion a clumsy process.

By contrast, the surface modeling requirements of Cartesian methods are substantially less intensive. If one works directly with CAD or NURBS, libraries exist for clipping Cartesian cells against these data (DTNURBS[35], ParaSolids[65], etc.). Alternatively, one may triangulate the components of a configuration independently with a pre-processor. Since components are free to intersect and overlap, there is no need to constrain component triangulations to intersection curves. In addition there is no need for length scales on neighboring component triangulations to match since the surface description is de-coupled from the volume mesh. Component motion or deformation may then be pre-programmed which permits macroscopic control of several cases without user intervention.

1.3.4 Asymptotic Complexity

One final attractive aspect of Cartesian approaches is that mesh generation time can scale linearly with the number of cells for a given configuration. Figure 1-4 shows a scatter plot of number of cells vs. CPU time for a series of meshes surrounding a double teardrop configuration. The solid line fitted to the data is the result of a linear regression. The line has a slope of 4.01×10^{-5} seconds/cell and a correlation coefficient of 0.9997. This equates to about 24950 cells/second or 1.5Mcells/minute on a desktop workstation (RISC R10000 at 195Mhz). The strong correlation to a straight line demonstrates that the mesh generator produces cells with linear asymptotic complexity, which is optimal for any method that operates cell-by-cell.

The result in Figure 1-4 is not necessarily surprising since Cartesian grids have a prescribed connectivity which is defined using only local data. Site (vertex) locations are predetermined by the bounding-box of the domain and the allowable number of refinements.

The result of linear complexity ($O(N)$) compares favorably with the asymptotic performance of popular 3-D Delaunay approaches for generating unstructured tetrahedra. The algorithms of Bowyer[22] and Watson[90] have best case complexity of $O(N^{5/3})$ and $O(N^{4/3})$ respectively. In reference [9] Barth argues that since the maximum numbers of tetrahedra that can be generated from N sites is N^2, these methods have a worse case bound of $O(N^2)$. Typical real world running times for efficiently written Delaunay algorithms fall between these bounds.

1.4 Cut-Cell Cartesian Methods

Before generalization to curvilinear coordinates, the first efforts in CFD relied on structured Cartesian methods. Most early approaches focused on the accuracy of spatial operators, and the efficiency of time advancement schemes. Typically, they did not allow mesh embedding and frequently used a “staircased” description of any geometry in the flow. By comparison with these early implementations, the advent of curvilinear meshes represented a major advancement in the accuracy of CFD meth-

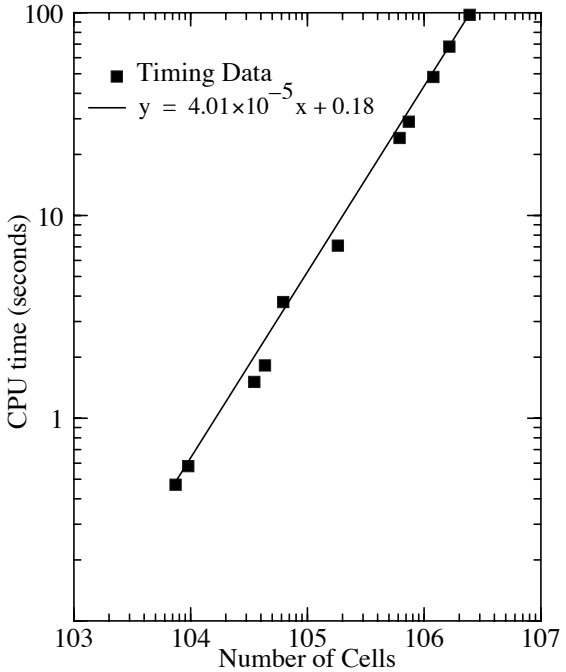


Figure 1-4: Scatter plot of mesh size vs. computation time for a double-teardrop configuration. RISC R10000 CPU, 195Mhz.

ods. While some commercial software still uses these non-adapted Cartesian grids with staircased geometry (often aimed at “under the hood” automotive applications) the focus of Cartesian technology development in the past decade has been almost exclusively on adaptive techniques with cut-cell boundaries. The “Cartesian methods” referred to by these notes belong exclusively to this latter class of approaches. It is precisely the ability to change mesh scale through adaptation and preserve boundary fidelity through cut-cell implementations which make Cartesian methods competitive with block-structured and unstructured (tetrahedron-based) methods. This section presents a brief overview of various Cartesian approaches being actively developed within the research community.

1.4.1 Structured Mesh Cartesian Approaches

Most modern structured Cartesian approaches permit local mesh adaptation through the use of embedded grid patches. The most popular structured approach is currently the Adaptive Mesh Refinement (AMR) strategy pioneered by Berger[19,14]. Since its development AMR has been successfully applied to the Navier-Stokes equations, reacting flow problems, incompressible flows, time dependent problems and a variety of other equation sets[5,66,6,59,74]. The technique has also been applied on curvilinear meshes with embedded patches[15,68]. This presentation follows closely that in reference [13].

In essence, AMR is a hierarchical inter-mesh communication scheme. It relies on block-structured, locally refined mesh patches to increase the resolution of an underlying structured coarse grid. Mesh patches at different levels are stored in a tree-based hierarchy. The method begins with an estimate of the local truncation error within each cell of a coarse grid patch. Cells with high error are then tagged for later refinement. Rather than divide each individual cell, however, the tagged cells are organized into rectangular grid patches which usually contain hundreds to thousands of cells per patch. The 2-D example in Figure 1-5[1] illustrates this blocking process on an example where a set of tagged cells around a circle are organized into 8 refined grid patches.

Since it attempts to create rectangular regions, the blocking process necessarily includes some cells not tagged for refinement. Thus, its natural to examine the efficiency of the blocking strategy. Typically 70% of cells in grid patches are tagged, this implies that the blocking process results in an “overhead” of 30%. Nevertheless,

1. The sketch in Figure 1-5 is modeled after [76].

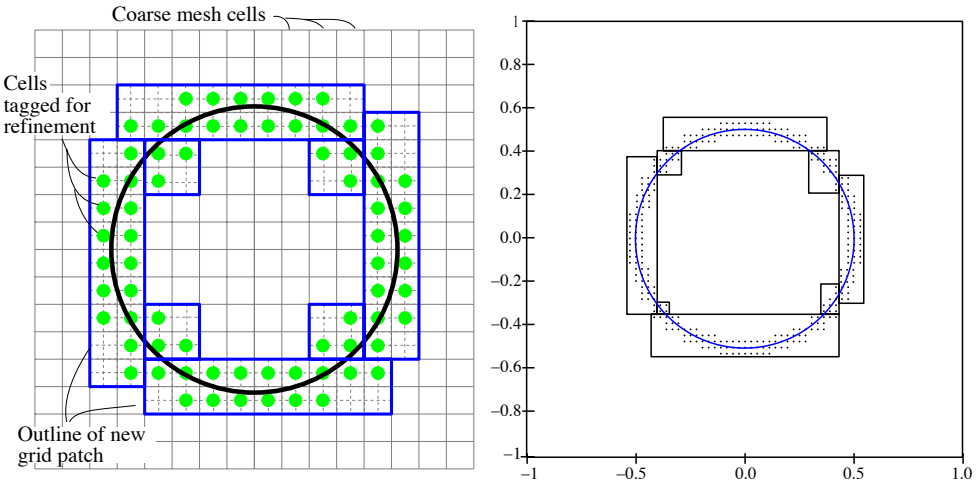


Figure 1-5: AMR mesh blocking for cells tagged around a circular region in 2D. Left: Schematic of cells tagged for refinement with patch outlines indicated by heavy lines. Right: sample calculation from Reference [76]

advocates of the approach are quick to point out that entire mesh patches may be completely specified with less than 20 words of storage per grid. Thus, total memory is still more compact than the 30-50 words-per-cell typical of unstructured or octree based adaptive schemes. When one considers that a typical three dimensional grid system may have thousands of embedded patches, the memory efficiency of AMR becomes obvious. For example, all cell geometry in a domain with 2^{10}patches would require only 80Kb (with 32-bit data), and may uniquely describe millions of mesh cells. This is far less memory than even the most efficient unstructured (cell-by-cell) approaches described in the literature[2].

AMR for Transient Flow Simulations

AMR has been used extensively in transient flow computations. In the solution of time dependent problems, AMR easily permits sub-cycling in time on finer mesh patches. It also allows grids to refine and coarsen without modifying the kinds of cumbersome 1-D cell arrays or connectivity trees common in octree or unstructured Cartesian approaches.

An AMR scheme for time dependent flows may be sketched as follows:

1. time step controller and integration
2. inter-grid communication
   2.a. Provide boundary conditions to interior grids
   2.b. Ensure conservation at patch interfaces
   2.c. Initialize solution on new grid patches
3. Cell-based truncation error estimation
4. Organize tagged cells into rectangular blocks
5. Generate new patches
6. Return to 1

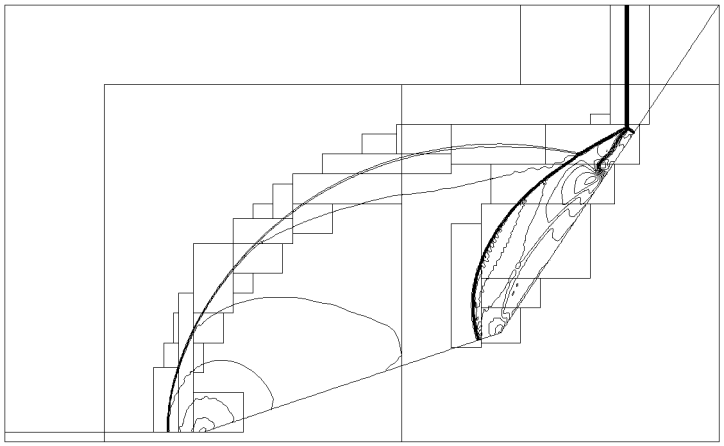


Figure 1-6: Density contours for a time dependent shock reflection from double ramp using AMR on Cartesian meshes, mesh patch outlines are shown. (reprinted from [13] with permission).

Figure 1-6 shows an example of a 2-D time dependent result for a shock reflection problem from a double ramp using AMR.

Error Estimation for Adaptation with AMR

Since they operate on a series of embedded meshes, AMR approaches lend themselves to direct truncation error estimates for controlling mesh adaptation. The possibility of performing direct Richardson extrapolation based error estimation is particularly attractive since it offers firm mathematical basis and thus avoids some of the ambiguities associated with many of the more *ad-hoc* "feature detection" type schemes discussed in the literature.

The overview presented below assumes that the method has the same nominal order of accuracy in both space and time. This restriction can be lifted by treating spatial and temporal operators separately.

Consider again the scalar advection equation defined on a unit domain in 1-D (eq. 1.1). Centered spatial differencing with Lax-Wendroff time advancement yields a scheme with second order accuracy in both space and time.

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{2h}(u_{i+1}^n - u_{i-1}^n) + \frac{\Delta t^2}{2h^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) = Q_h u_i^n \qquad (1.8)$$

u_i^n is the discrete approximation to the exact solution $u(x,t)$ at $x_i = i \cdot h$ and time $t^n = n \cdot \Delta t$. The local truncation error at time t^{n+1} is then:

$$u(x_i, t^{n+1}) - Q_h u(x_i, t^n) = \frac{\Delta t^3}{6} u_{ttt} + \frac{\Delta t h^2}{6} u_{xxx} + \ldots \tag{1.9}$$

Assuming $\Delta t = Const \cdot h$ yields spatial and temporal error of the same order. After two time steps, the leading term of the truncation error expression is doubled.

$$u(x_i, t^{n+2}) - Q_h Q_h u(x_i, t^n) = 2 \cdot \left(\frac{\Delta t^3}{6} u_{ttt} + \frac{\Delta t h^2}{6} u_{xxx} + \ldots \right) \tag{1.10}$$

Equation (1.10) for the local truncation error may be re-computed on a coarser mesh created by removal of every other grid point in the fine mesh. Defining $x_{i+1/2} \equiv \frac{x_i + x_{i+1}}{2}$ and $u_{i+1/2} \equiv \frac{u_i + u_{i+1}}{2}$ permits us to define Q_{2h} as the discrete Lax-Wendroff operator on the coarsened mesh. Assuming $\Delta t/h$ remains the same as on the fine grid, the local truncation error of the coarser mesh is:

$$u(x_{i+1/2}, t^{n+2}) - Q_{2h} u(x_{i+1/2}, t^n) = 8 \cdot \left(\frac{\Delta t^3}{6} u_{ttt} + \frac{\Delta t h^2}{6} u_{xxx} + \ldots \right) \tag{1.11}$$

Comparing the average discrete solution at x_i and x_{i+1} after two time steps $(Q_h^2 u(x_i, t^n) + Q_h^2 u(x_{i+1}, t^n))/2$ with the solution on the coarse mesh after one time step yields a difference which is proportional to the leading term in the truncation error expression on the fine mesh.

$$\frac{(Q_h^2 u(x_i, t^n) + Q_h^2 u(x_{i+1}, t^n))}{2} - Q_{2h} u(x_{i+1/2}, t^n) \approx 6 \cdot \left(\frac{\Delta t^3}{6} u_{ttt} + \frac{\Delta t h^2}{6} u_{xxx} \right) \tag{1.12}$$

In eq. 1.12 the difference of discrete operators on the left produce a result which is precisely proportional to the leading terms on the right side of eq. 1.9, thus, by comparing solutions on the two telescoping meshes we have a Richardson type measure of the actual truncation error.

The second term on the left of eq. 1.12 can be obtained by calling the integrator for one step on the coarse grid, similarly the Q_h operator is evaluated on the actual Cartesian mesh itself. Cells with truncation error measures above some (statistically determined) threshold may then be tagged for refinement as in Figure 1-5.

The Taylor expansions in eq. 1.9 imply that this estimator is actually only appropriate for smooth data. The derivation has also been extended for application to solutions with discontinuities. Reference [13] presents an overview of this extension.

1.4.2 Octree/Quadtree Based Cartesian Methods

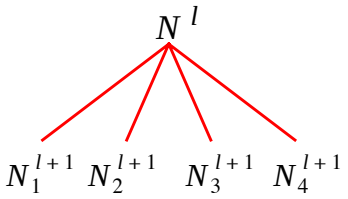
Tree-based data structures have long been a mainstay of hierarchical storage systems in computer science. Binary trees were originally created for data sorted on a single key, and are examples of *linear data structures*. While 1-D data (like a dictionary) may be maintained in a typical binary tree, spatial data in higher dimensions requires searching on multiple keys. Quadtrees and octrees are two examples of *spatial data structures*[1]. *A* variety of classic texts discuss efficient algorithms for maintaining multi-keyed data with an implied hierarchical relationship[49,77,37].

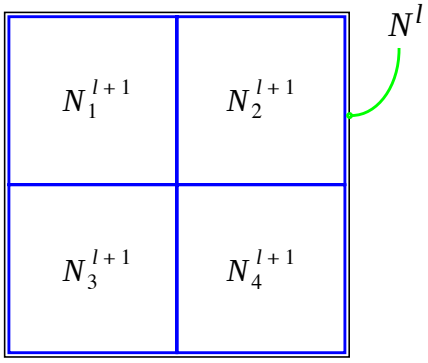
The nested nature of *h*-refined Cartesian meshes makes them good candidates for these types of data structures. Two-to-one *h*-refinement produces two children in 1-D, four children in 2-D, and eight in 3-D (or 2^d in d dimensions). This gives natural mappings to binary, quad- and octree approaches for Cartesian meshes. Mesh connectivity information is implied by a tree's logical structure. Additionally, the geometry of a Cartesian cell is completely specified by its location in the tree and the geometry of the root cell. Thus, these data structures can provide not only local mesh topology, but also cell geometry as well.

Terminology

Tree structures mimic more the form of "family trees" than that of trees in nature. In the most fundamental implementation, a regional quad- or octree starts from a single *root* node which covers the entire computational domain. The actual cells in a Cartesian mesh make up the *leaves* of the tree. The term *node* generically refers to any storage location in the tree (root, leaf, etc.). Continuing with this metaphor, genealogical terms are used to characterize the relationship between cells at various levels of refinement within an adaptive grid. For example, *h*-refinement of a cell in a 3D mesh creates eight *children*. Each of these usually retains a pointer (or *link*) back to its *parent*. Similarly, parent nodes contain links to the addresses of their children. These pointers usually follow some fixed ordering scheme so that a child's position is

1. A partial list of useful spatial data structures includes (at least); binary trees, *k-d* trees, point optimized quad/octrees, regional quad/octrees, bucket quad/octrees, alternating digital trees, dynamically quantized pyramids, range-trees, *kdb*-trees, *R*-trees, *hB*-trees, *MX*-trees, bin-trees, binary-space-partitioning trees, polygon trees, sector trees, 2^n-trees, sequential and inverted lists, linear and spiral hashtables, priority search trees, and a variety of *PR*-trees, see Ref.[77].

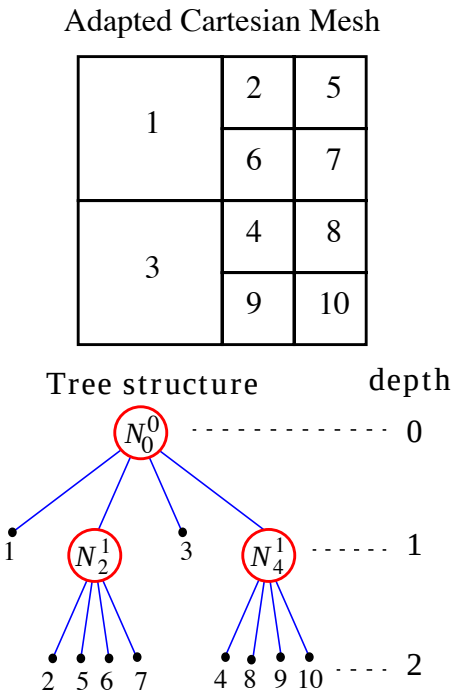




Address	Node	Parent	Child 1	Child 2	Child 3	Child 4
A_0	N_0^0	0	A_1	A_2	A_3	A_4
A_1	N_1^1	A_0	0	0	0	0
A_2	N_2^1	A_0	0	0	0	0
A_3	N_3^1	A_0	0	0	0	0
A_4	N_4^1	A_0	0	0	0	0

Figure 1-7: Parent-child relationship in a Cartesian mesh using a regional quadtree data structure. Shown also is the corresponding sub-tree and a simple scheme for array based storage.

uniquely specified with respect to its siblings. Figure 1-7 shows one example of such an ordering for a single parent node and its four children in a 2-D quadtree mesh. The quadtree shown here is properly referred to as a "regional quadtree", since each node in the tree corresponds to a region in the domain[77]. Each node in the tree may be pointed to by only one parent node, and only the root has no parent. Leaf nodes



Address	Node	Parent	Child 1	Child 2	Child 3	Child 4
A_0	N_0^0	0	A_1	A_2	A_3	A_4
A_1	1	A_0	0	0	0	0
A_2	N_2^1	A_0	A_5	A_6	A_7	A_8
A_3	3	A_0	0	0	0	0
A_4	N_4^1	A_0	A_9	A_{10}	A_{11}	A_{12}
A_5	2	A_2	0	0	0	0
A_6	5	A_2	0	0	0	0
A_7	6	A_2	0	0	0	0
A_8	7	A_2	0	0	0	0
A_9	4	A_4	0	0	0	0
A_{10}	8	A_4	0	0	0	0
A_{11}	9	A_4	0	0	0	0
A_{12}	10	A_4	0	0	0	0

Figure 1-8: Quadtree mesh, regional quadtree structure and simple array based storage scheme.

are distinguished by the fact that they have no children. Figure 1-7 also shows a simple 1-D array-based storage scheme for mapping a regional quadtree tree to memory. Many programming languages with flexible datatypes permit more elegant representations of tree structures but the simple scheme shown is sufficient for this discussion.

Figure 1-8 presents a slightly more complex example for a quadtree mesh with three levels of tree nodes and 10 leaf nodes for the cells in the computational domain. In this example, tree nodes are identified with the notation N_C^l where l is the depth of the node in the tree and c is the child number of a node on its parent's child list. The addresses in this example are sorted by depth for clarity, but since the address of the children are stored directly, the tree can be kept in any order (including many sorted orders which can be optimized to improve instruction cache performance on cache-based CPU's). We note that the geometry of the root and a node's logical position in the tree uniquely define each Cartesian cell in the domain

Connectivity Queries

One of a tree's main functions is to provide information about a cell's logical relationship to other cells in the domain. As an example, consider the implementation of a cell-centered flow solver on the 2-D mesh shown in Figure 1-9. Cell-centered storage of state data implies that each cell will exchange flux with its nearest face-neighbors. The figure displays the tree-traversal paths for locating the north face neighbors of cells 1, 3, and 5. Notice that while cell 1 need only query its parent to get to its north neighbor (cell 2), cell 3 must return all the way back to the root node to find cell 4 across its north face.

This example shows both the strengths and weaknesses inherent to tree-based approaches. The tree structure is relatively compact, and provides virtually all mesh connectivity information while simultaneously permitting direct computation of geometric properties. Geometry and cell size need not be stored explicitly. In addition, adaptation is relatively easy to implement since the structure is extensible through division of leaf nodes. However, this compactness and flexibility come at the expense of tree traversal overhead.In an effort to reduce this overhead, most tree-based methods use some local neighborhood storage for leaf nodes. For example, one may pre-compute all the distance-one neighbors to a mesh cell to avoid the look-up overhead associated with the connectivity queries in Figure 1-9. This then becomes a case of trading off execution speed for memory usage. Reference [70] reports that 10-

20% of the CPU time for a 2-D quadtree-based flow solvers is dedicated to tree traversal. Typical implementations of Cartesian flow solvers on 3-D octree meshes usually store 30-50 words per cell.

Figures 1-10 and 1-11 display flow examples computed with octree and quadtree meshes. Figure 1-10 shows surface isobars and mesh-cuts from a business jet configuration computed by Charlton and Powell[25]. The mesh-cut through the aircraft symmetry plane provides an indication of the mesh resolution used in this simulation. This configuration was computed at $\alpha = 0°$, and $M_\infty = 0.87$. The complete mesh around the aircraft consisted of 583000 cells, among which 182000 actually intersected the wetted surface of the geometry.

Figure 1-11 shows an unsteady 2-D result computed on a quadtree mesh[70,12]. Several snapshots from the time history of the computation are included. The example illustrates the progress of a jet-powered projectile penetrating a deformable shell structure in a quiescent stream. In this example, a new mesh is generated at subse-

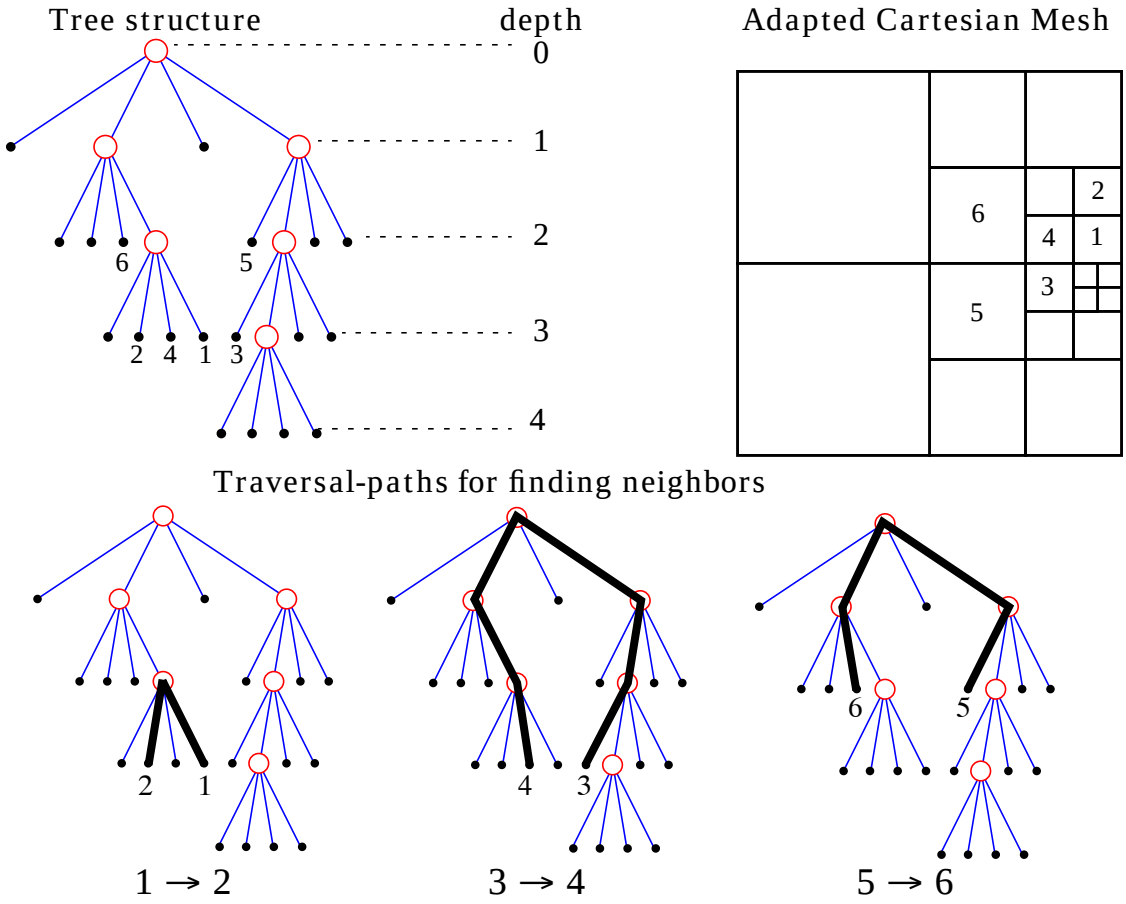


Figure 1-9: Tree-traversal paths for locating the north face-neighbors of cells 1, 3 and 5.

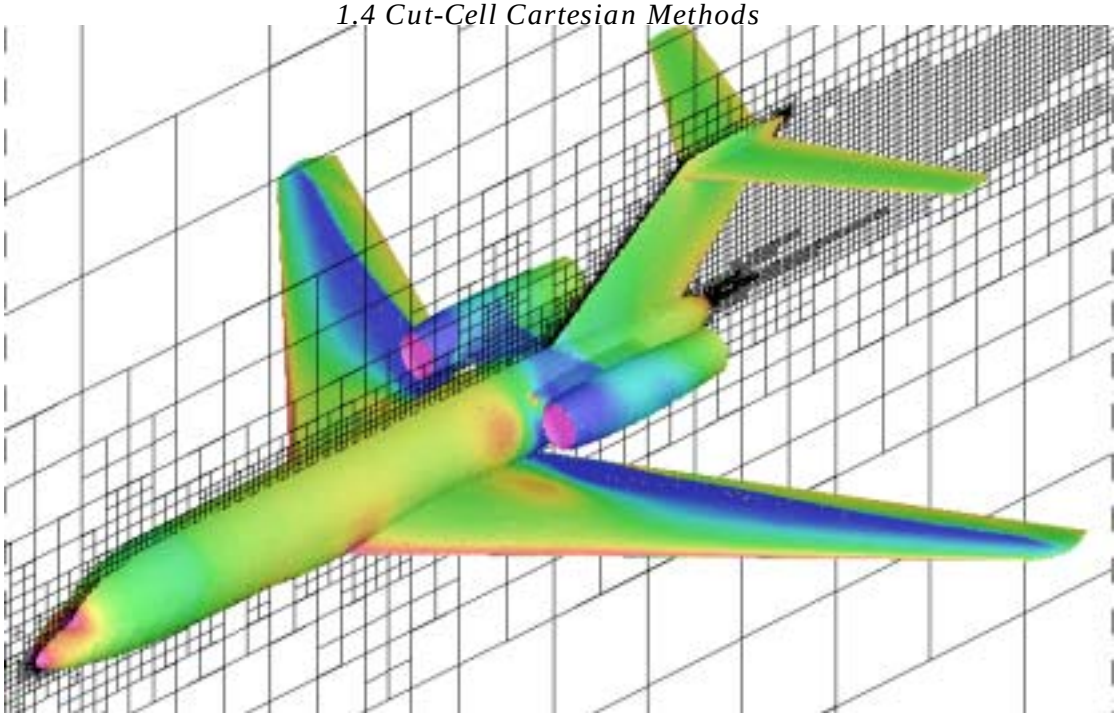


Figure 1-10: Isobars and mesh cuts on a business jet configuration computed with an octree based approach (reprinted from Ref. [25] with permission).

quent time steps to track the progress of the geometry and flow as both the body motion and flow evolves.

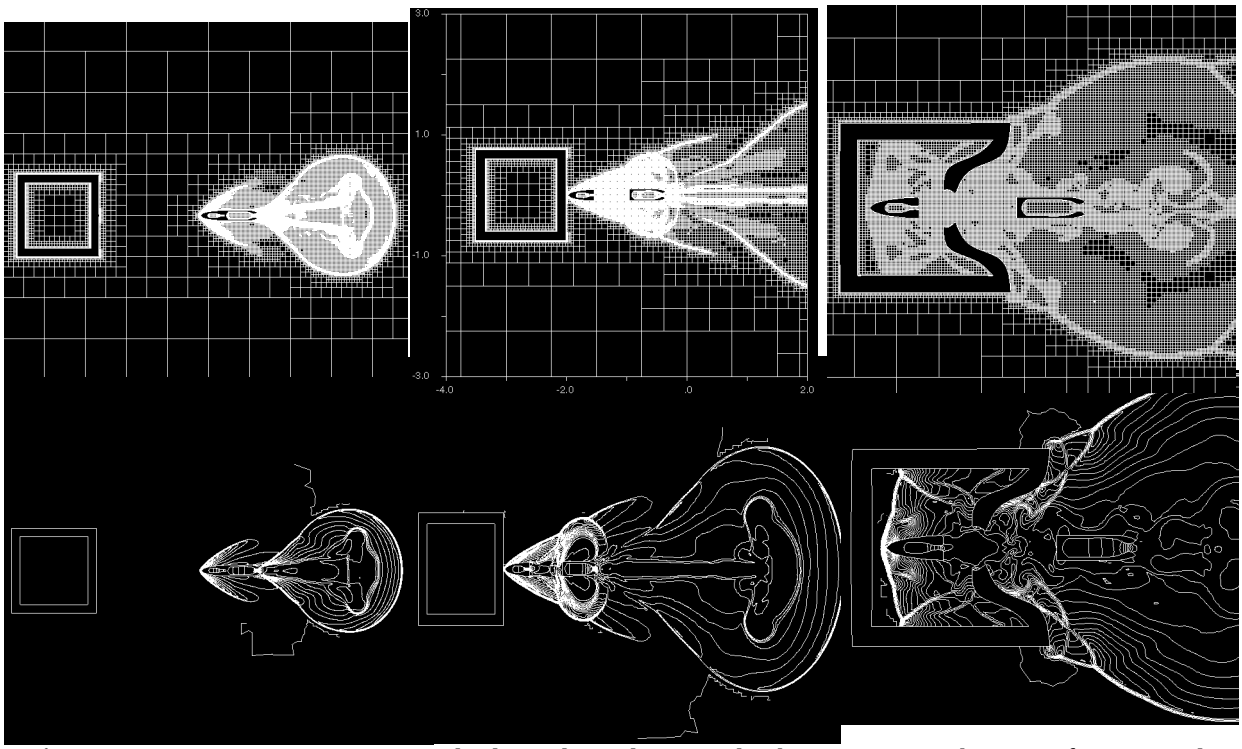


Figure 1-11: Density contours and adapted quadtree grids showing a time history of a projectile penetration problem (reprinted from Ref. [70] with permission).

1.4.3 Unstructured Cartesian Approaches

Despite the apparent oxymoron, many Cartesian implementations rely on fully unstructured data structures. In this context, "unstructured" refers to the fact that hierarchical information is not used to infer mesh topology. Instead, connectivity is explicitly stored. Data structures from finite element methods or Computational Geometry are employed in much the same way that they are for unstructured approaches on tetrahedral meshes. Such structures provide more flexibility than patch-based or even tree-based architectures since they easily incorporate the possibility of anisotropic (directional) refinement of Cartesian cells.

Figure 1-12 shows a few examples of directional refined Cartesian cells. Anisotropic refinement permits 1:2 or 1:4 division of hexahedral cells, and permits local modification of the cell aspect ratio in response to flow features or geometry. "Counting arguments" brought up by recent research with large computations indicate that anisotropic cell division can become extremely important in three dimensions[3]. This is especially so when analyzing geometries with high aspect ratio components (*e.g.* flaps, flap-vanes, spoilers, etc.). Some such results will be discussed in Section 3 of these notes.

Data Structures

A variety of local data structures for describing mesh connectivity exist in the literature[11,9]. Figure 1-13 depicts a few of the more common ones used in CFD and computational geometry. Vertices in the domain are generally assumed to be uniquely defined, and the structures "point" to vertices either directly (using pointers) or by an indexing scheme.

One of the most frequently encountered structures lists the cell-to-vertex connectivity of an element (Fig. 1-13.a). This is usually referred to as the "standard" finite-element structure. A variation of this is to list the cell-to-edge (or cell-to-face)

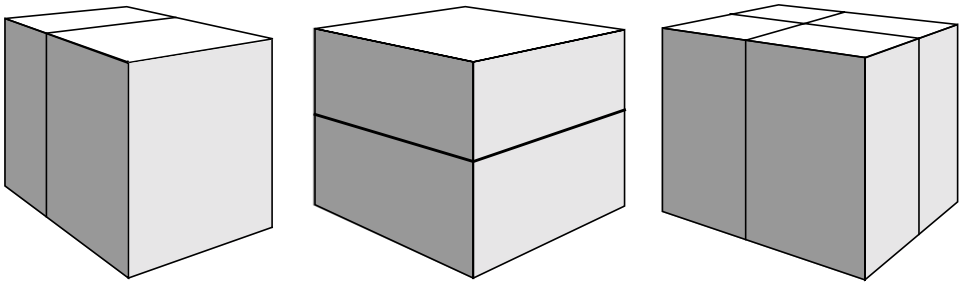
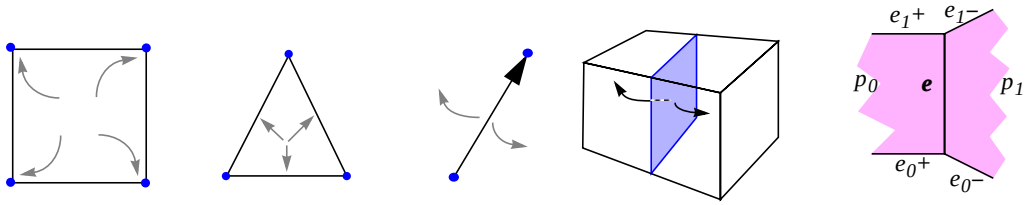


Figure 1-12: Anisotropic (directional) division of Cartesian cells.



a. cell-to-vertex b. cell-to-edge c. edge d. face e. winged-edge

Figure 1-13: Some common data structures for polyhedral tessellations in two and three

connections as shown in 1-13.b. This structure is often found in planar or manifold triangulation schemes, and is sometimes used in tetrahedral mesh generators. These two structures tell an element about itself, that is, they provide communication *within* an element, or about the *extent* of an element (D_0 communication). By themselves, they provide no information about mesh connectivity. They can, however, be used for *scatter* and *accumulate* operations of the type required in cell-vertex CFD schemes like that found in reference [62][1].

Many planar graph algorithms make use of the edge structure like that shown schematically in frame (c) of Figure 1-13. The edges may be directed (as in the sketch) and may include links to the cells on either side. This structure is fundamentally different from the element-based structures since it straddles mesh elements. It therefore provides a method for direct communication between two adjacent (D_1) elements. When the cell-to-edge structure is used in combination with an edge structure, one can rapidly traverse a local neighborhood on an unstructured mesh. Since it lists an edge's origin and destination vertices, the edge structure also provides nearest-neighbor connectivity on the *dual* of a planar graph. Mesh duals are a common construct in node-based CFD algorithms where control volumes are constructed around the vertices of the physical mesh (see for ex. Ref.[10]). Researchers have exploited the local connectivity offered by edge structures to show that various discrete operators may be cast as a series of edge-based operations. Reference [9] reviews the use of edge formulas for constructing discrete differentiation, Laplacian, and Hessian operators using dual control volumes on a planar graph.

For 3-D cell-centered schemes, the face structure (Fig.1-13.d) provides much of the same functionality as the edge structure gives to node-based methods. This relation-

1. In cell-vertex schemes, updates are computed within each mesh cell, these updates are then scattered to the cell's vertices using a distribution process. When the cell loop is complete, a complete set of pointwise updates have accumulated at the mesh vertices.

ship is strongly linked to the property of duality. If one constructs a dual mesh by connecting the nearest neighbor centroids of a 3-D arrangement of Cartesian cells, there is a one-to-one correspondence between edges in the dual and faces in the original mesh. Every edge in the dual pierces one face of the original mesh. Thus, the edge structure for the dual is precisely a face structure for a cell-centered Cartesian mesh.

Another popular data structure for traversing unstructured meshes is the winged-edge structure in Figure 1-13.e. Originally developed by Baumgart[11] for work in computer vision, the structure links an edge with its origin and destination vertices, the two polygons on either side of the edge, and four edges $e_0\pm$ and $e_1\pm$. This structure is particularly useful for storing polygonal faces of unknown degree, since one may start at any edge on a polygon, and traverse around its edge following the $e+$ or $e-$ loops. When a 3-D Cartesian cell intersects the surface triangulation of a body, its cut faces constitute polygons of an unknown degree. The winged-edge structure provides a simple way for traversing the perimeter of these cut faces.

The memory requirements for unstructured data storage are generally higher than that for quad/octree approaches. Typical implementations use about 30-100 words-per-cell. However, recent research results have demonstrated that this is not necessarily the case. By tailoring unstructured formats for Cartesian meshes, memory usage can be substantially reduced. For example, the mesh generator in reference [2] uses only about 9 words-per-cell (neglecting storage of boundary geometry). Section 3 of these notes presents a compact data structure for storing all geometry and cell-to-vertex pointers in 96 bits of memory (3 words on 32-bit architectures).

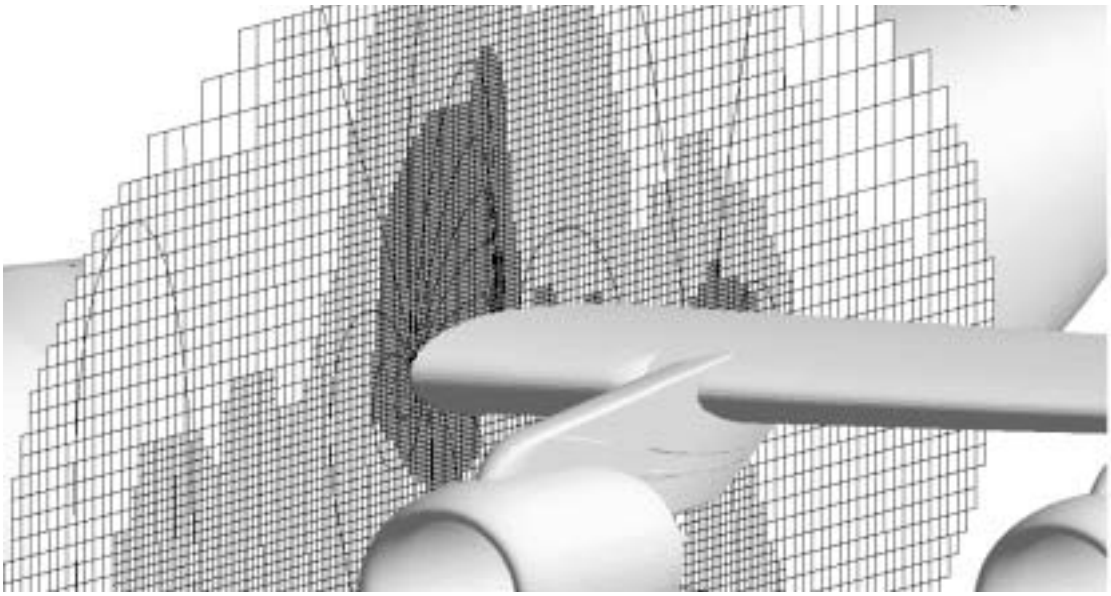


Figure 1-15: Adapted mesh, and computed isobars for inviscid flow over a High Wing Transport (HWT) configuration. The unstructured Cartesian mesh contained 2.9M cells with 10 adaptations.

Results from Unstructured Implementations

Results using unstructured Cartesian mesh approaches are widely reported in the literature[58,3,2,56]. Figure 1-14 shows the adapted Cartesian mesh and computed

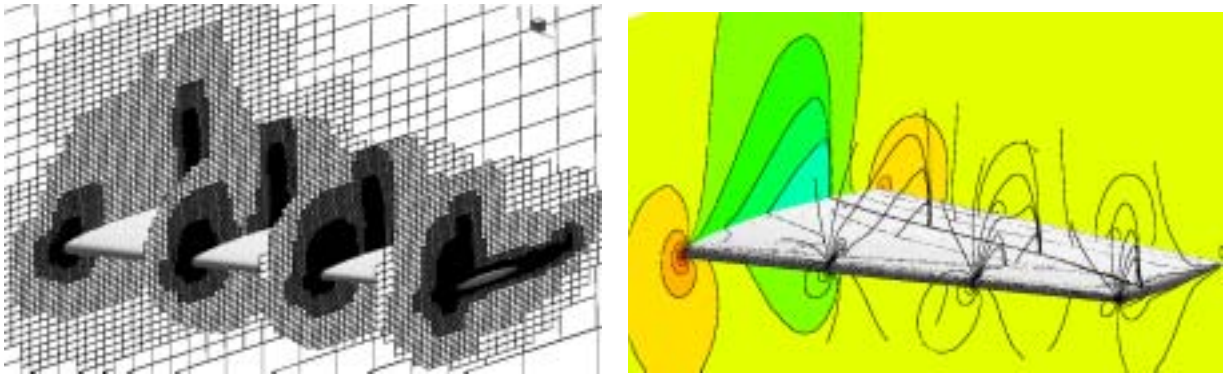


Figure 1-14: Adapted mesh, and computed isobars for inviscid flow over an ONERA M6 wing at $\alpha = 3.06°$, and $M_\infty = 0.84$, computed using an unstructured representation of the Cartesian mesh.

isobars for an ONERA M6 wing example at $\alpha = 3.06°$, and $M_\infty = 0.84$. The mesh shown contains approximately 1.2M cells and was generated by 9 successive *h*-refinements of an initial coarse grid.

The High Wing Transport (HWT) example shown in Figure 1-15 contains an adapted Cartesian mesh over a complete transonic transport aircraft. The mesh in this exam-

ple contains 2.9M cells, and the figure shows a close-up of the flow structure between the fuselage and inboard nacelle.

The two examples in Figs.1-14 and 1-15 combine unstructured data storage with a component-based surface modeling for complete aircraft geometries. Surface triangulations are used to describe the component geometry as discussed in §1.3.3. These component triangulations were generated from CAD and are permitted to overlap and intersect. The result in Figure 1-16 uses this same approach to model a HWT with its high-lift system deployed resulting in a total of 18 components. The component triangulations used in this example describe the geometry using a total of over 700000 triangles, and 16 of the components intersect. The mesh shown (at selected cuts) includes 1.65M cells and the inset frame shows the resolution available through the flap system. In all, 10 levels of cells comprise the final mesh. Adaptation in this example was triggered by a simple criterion examining the undivided first difference of density. At a low subsonic Mach number and a moderate angle of attack, this indicator responded primarily to the suction peaks on the leading-edge-slat and main element, as well as the inviscid jet through the flap system. Despite the fact

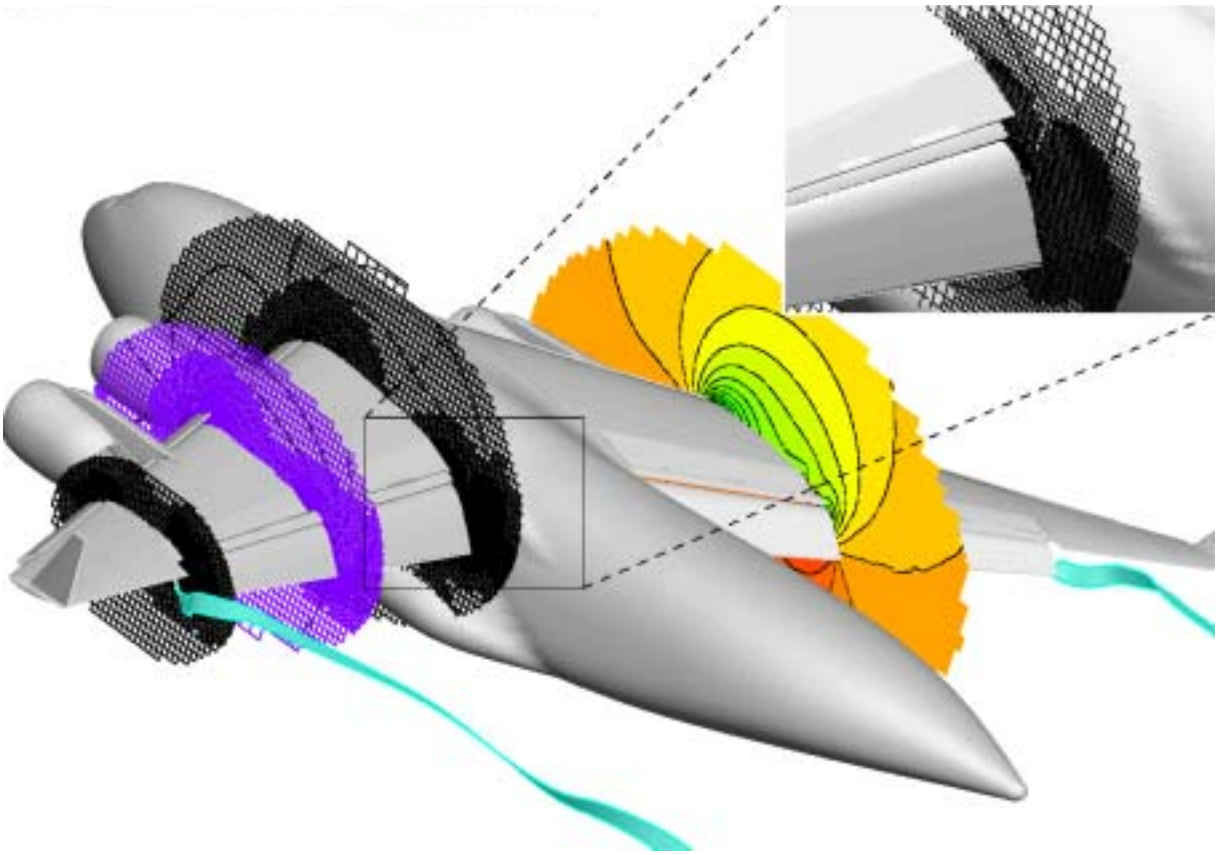


Figure 1-16: HWT example with high-lift system deployed. The mesh contains 1.65M cells at 10 levels of refinement. The mesh is presented by cutting planes at 3 spanwise locations and the cutting plane on the starboard wing is flooded by isobars of the discrete solution.

that this simulation is inviscid, the sharp outboard corner of the flap has correctly spawned a flap vortex which is evidenced by a stream ribbon in the figure.

1.5 Rapid Mesh Traversal and Searching

Having completed a brief overview of various approaches for Cartesian methods, this section concludes by presenting some geometric algorithms which are central to the efficient implementation of Cartesian methods aimed at realistically complex geometry. Subsequent sections of these notes make extensive use of the algorithms in this section for searching and mesh traversal. This infrastructure is important since a clumsy approach to these fundamental operations will seriously degrade the asymptotic performance of many of the algorithms presented later.

A component-based description of a configuration specifies the complete geometry as the set of all its boundary components, B_k. For a configuration Ω, with K components, this relationship may be formalized by:

$$\Omega = \{B_1, \ldots, B_k, \ldots, B_K\} \tag{1.13}$$

The individual components in Ω may come from a variety of sources, including CAD, STL data, IGES[75] descriptions, etc. In all cases, it's likely that each component is itself described by a collection of geometric primitives (*e.g.* CAD data for a fuselage may be a collection of trimmed NURBS, a wing may be specified as triangulated STL data etc.). Thus, the k^{th} component, B_k, will consist of n_k primitive objects. N, the total number of primitive objects in the configuration is then:

$$N = \sum_{k=1}^{K} n_k \tag{1.14}$$

If this configuration is contained by a Cartesian mesh, let M denote the total number of hexahedral cells which actually intersect the wetted surface of the geometry.

In generating the mesh, and setting up numerical boundary conditions, each cut cell needs to be intersected against the configuration. There are M cut-cells, and N objects in the configuration. Thus, without special care, the computational complexity of the intersection operation will be $O(N \cdot M)$. Since both the surface geometry and adaptive grids typically cluster cells near rapidly varying geometry, N and M are generally of the same magnitude. Thus, this result implies quadratic complexity.

The HWT configuration in Figure 1-16 had 700000 triangles describing the geometry, and just over 600000 cut Cartesian cells in the computational mesh. Clearly, an $O(N \cdot M)$ exhaustive test on problems of this magnitude would be out of the question. What is required is an operator with better asymptotic performance, which returns only the short list of triangles that can potentially intersect each target cut-cell. Since the objects returned by this operator must be geometrically close to a given target cell, the process of forming this list is referred to as a *proximity query*.

Efficient proximity testing is an important part of the infrastructure in the design of any algorithm targeted at complex geometry and high-resolution. Such problems will necessarily contain many geometric components and many cells in the computational domain, thus good asymptotic behavior is a primary concern. This section reviews a spatial data structure which is particularly well suited to the task of rapidly locating finite sized objects in the neighborhood of a given target. Moreover, it is general enough to equivalently handle a variety of primitive object types, including triangles, NURBS and other entities.

A second important algorithm involves the efficient traversal of an unstructured collection of objects of the same type in order to propagate some property to a cell's local neighborhood. For example, in a Cartesian mesh, we may need to visit all Cartesian cells which are within the interior of a given component (to delete or mark them). Since the component may be irregularly shaped, it would be expensive to exhaustively check each Cartesian cell for containment. Instead, it may be more efficient to first find one cell which is inside and then quickly mark that cell's neighbors with the same local property (*e.g.* "is inside"), provided that they pass some simple local test (like "is not intersected"). Such *painting algorithms* are immensely powerful in reducing the number of expensive tests required to classify a long list of objects, since they quickly propagate a result to a region of the mesh without having to independently test each cell.

Both of the algorithms presented in this section are general geometric tools with a myriad of uses outside the specific topic of Cartesian methods. They are common algorithms in computational geometry and are applicable to unstructured and structured mesh generation, finite element modeling, and domain decomposition.

1.5.1 Proximity Queries

The Alternating Digital Tree (ADT)[20] is a spatial data structure that is particularly well suited to searching (possibly) heterogeneous collections of objects. It attains this generality by abstracting each specific geometric entity with a set of uniquely defined keys which do not depend on the details of an object's construction. It uses these keys to position the object in a binary tree. Since the objects are stored using keys which are not simply physical space coordinates, the method may be classified as a hyperspace search technique[77].

Partitioning within the Search Space

Before examining how the tree is used to return an object list, first consider its construction using point data from a predetermined point set. An exhaustive search of N points is one which checks each point in the set against the search criteria. Obviously, such a procedure has linear complexity since searching through a set of $2N$ points will take twice as long. A host of techniques exist to search such a point set with $O(\log N)$ complexity. Included among these are the regional quad/octree structures from §1.4.2 and many others detailed in the literature[77,49,21,79]. However, the problem is considerably more complicated when one attempts to search over a set of discrete, finite-sized objects. For example, if one uses a regional quadtree to store two

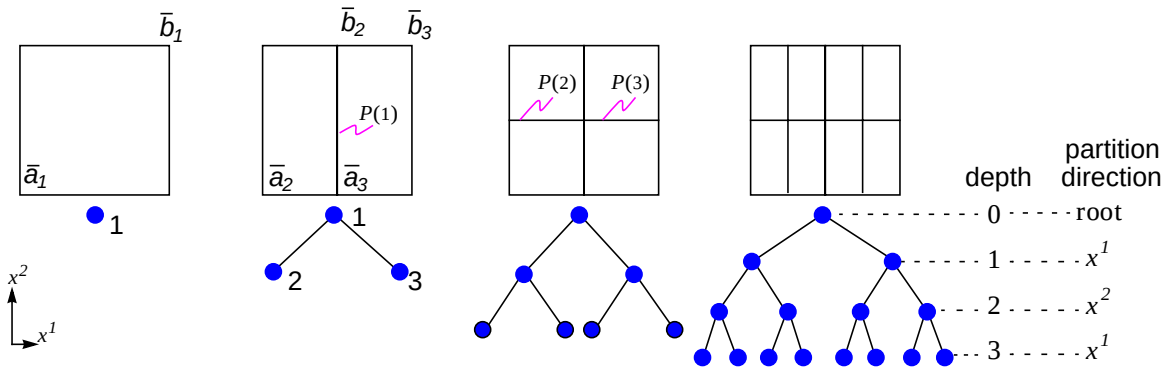


Figure 1-17: The relationship between partitions of the search space and levels of the binary ADT Partition directions alternate over the keys of the search space. (following Ref.[20])

dimensional objects, special procedures will be necessary to handle the cases where objects cross partition boundaries. The ADT was designed specifically for these types of searches.

Following reference [20], we introduce the ADT by examining its partitioning of a 2-D hyperspace defined by the first and second coordinates of a two-dimensional point set. If Q is the set of points to be searched, we can define the minimum and maximum position vectors, $\bar{a}_1$ and $\bar{b}_1$, of this set by a componentwise application of the *min* and *max* operators to all the points in the set. Using n as a running variable to sweep over the N elements in the set, and the superscript j to sweep over the spatial directions (here $d = 2$):

$$\begin{aligned} a_1^j &= min(Q_1^j, \ldots, Q_n^j, \ldots, Q_N^j) \qquad j \in \{1, \ldots, d\} \\ &\text{and} \\ b_1^j &= max(Q_1^j, \ldots, Q_n^j, \ldots, Q_N^j) \qquad j \in \{1, \ldots, d\} \end{aligned} \tag{1.15}$$

Since $[\bar{a}_1, \bar{b}_1]$ are the minimum and maximum coordinate limits of the entire data set Q, all the points in the set are contained by the region $[\bar{a}_1, \bar{b}_1]$ The subscript $(\,)_1$indicates that the region $[\bar{a}_1, \bar{b}_1]$ corresponds to tree-node 1. As shown in Figure 1-17, this region is the root of the binary tree which represents the ADT in a hyperspace spanned by the components of $\bar{a}_1$ and $\bar{b}_1$. We now begin bisecting this hyperspace with planes normal to the coordinate axes, alternating the direction of the partitioning plane at each level in the binary tree.

As a result of this cyclic bisection process, if a tree node k, is at a depth m in the tree, then its partition, $P(k)$ is perpendicular to the j^{th} coordinate axis, where j is given by:

$$j = 1 + m \,\%\, d \tag{1.16}$$

where the *mod* operator "%" takes the integer remainder of the integer division of its arguments, $m \div d$.

Just as the first node corresponds to the root of the tree, $[\bar{a}_1, \bar{b}_1]$, each subsequent subdivision uniquely identifies a region of the hyperspace with a node on the tree. In general, the k^{th} node corresponds to the region $[\bar{a}_k, \bar{b}_k]$ as shown in Figure 1-17. Subdividing this region with a partition normal to the x^j axis and located at $x^j = P(k)$ generates a child on the right $[\bar{a}_{k_R}, \bar{b}_{k_R}]$ and one on the left $[\bar{a}_{k_L}, \bar{b}_{k_L}]$. The regions covered by these children are determined by that of the parent node and the location of the partition plane. The components of the left child are[1]:

$$\begin{aligned} a^i_{k_L} &= a^i_k \\ b^i_{k_L} &= b^i_k \qquad \forall i \in \{1, \ldots, d\} \text{ and } (i \neq j) \end{aligned}$$

and when $i = j$,

$$\begin{aligned} a^j_{k_L} &= a^j_k \\ b^j_{k_L} &= P(k) \end{aligned} \tag{1.17}$$

Similarly, the components of the right child are:

$$\begin{aligned} a^i_{k_R} &= a^i_k \\ b^i_{k_R} &= b^i_k \qquad \forall (i, j \in \{1, \ldots, d\} \text{ and } (i \neq j)) \end{aligned}$$

and when $i = j$,

$$\begin{aligned} a^j_{k_R} &= P(k) \\ b^j_{k_R} &= b^i_k \end{aligned} \tag{1.18}$$

where j is determined by the depth of node k using eq. 1.16.

In its original form, partitions for the ADT were chosen by *geometrically* bisecting the point set and the resulting sub-regions[20]. For example, if the k^{th} node covers the region [0, 1] in the j^{th} direction, the partition $P(k)$ is placed at $x^j = 0.5$. However, since the points may not be evenly distributed, we have chosen to *logically* bisect the hypercube spanned by each node to improve the balance of the ADT. Thus, if a branch in the tree corresponds to a partition plane normal to the j^{th} axis, we sort the data in the node atop that branch using the j^{th} coordinate as a key. Then one may locate the partition at a position that corresponds to the median location in this sorted list, assigning half of the list to each child. This partitioning strategy ensures

1. Its helpful to refer to Figure 1-17 when following eqs.1.17 and 1.18.

that the left and right children of each node contain the same number of objects. Therefore, the tree remains balanced, even when the data is non-uniformly distributed in the search space.

As a consequence, this strategy implies that for each node in the tree, we must store the location of the partition associated with it. However, this additional word of storage is rewarded with the guarantee of a well balanced tree. The tree holds the maximum amount of data at every level, and is therefore not as deep as an unbalanced tree of the same number of objects. As a result, fewer operations are required when the tree is used to retrieve data. There is also slightly more set-up time since the data within each subregion must be sorted before the partition can be inserted. However, most of the uses for the ADT outlined in the following sections are for searching static data sets. Sorting overhead is a one-time penalty, and the gain in efficiency is substantial.

Data Storage and Retrieval

Thus far, this discussion has focused on partitioning the search space and showing the correspondence between regions in the search space and nodes in the ADT. The next step is to specify a method for associating elements in the set of objects, *Q*, with the nodes in the tree. The storage rule simply insists that for a node *k* to contain a point $\bar{x}_k$, the point must lie within the region covered by *k*.

$$a_k{}^i \le x_k{}^i < b_k{}^i \qquad \forall\, i \in \{1, \ldots, d\} \tag{1.19}$$

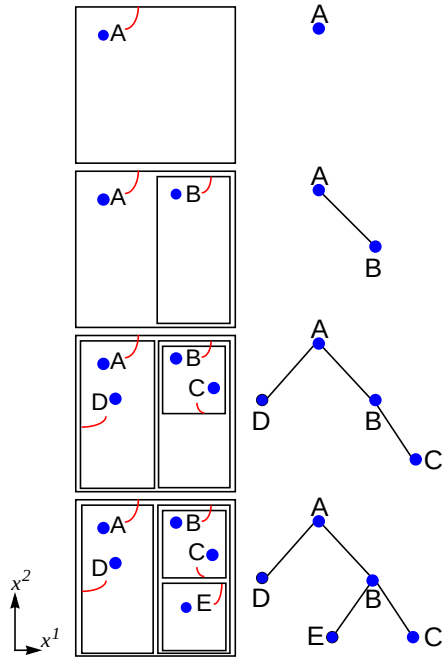


Figure 1-18: Construction of an ADT for a data set with 5 points in a two dimensional hyperspace. (following Ref.[20])

The *N* objects in *Q* are added to the tree following the scheme outlined by eq. 1.19. Additionally, each node of the tree is itself associated with one of the objects in its region (this way, the tree stores data in both its branches and its leaves). Figure 1-18 shows an example of an ADT built for data with two search keys. Each point in the data set is linked to the tree node which is used to store it. In this figure, point *A* is associated with the root, and point *B* is associated with *A*'s right child at a tree depth *m = 1*. Notice that the

"left" and "right" children of a node correspond to the regions which are lower and higher (respectively) in the coordinate by which that node is partitioned. Nodes continue to be added to the tree until all the data in a subtree has been uniquely associated with a node.

With the partitions in place and the data linked to the tree's nodes, one may now outline an algorithm for returning a list of objects which falls within a specified target range. Note that this is precisely the operator which we sought at the introduction to this section.

Denoting the target region by its minimum and maximum coordinates $[\overline{T}_{min}, \overline{T}_{max}]$ we want to return the list of all objects in Q which are contained in the target region. In this algorithm, subtrees are identified by the name of their root nodes, the left and right children of subtree S are denoted by S_L and S_R, and the coordinate data for the point stored in node S is $\overline{x}_S$.

In essence, we seek the intersection between the target region $[\overline{T}_{min}, \overline{T}_{max}]$ and the set of objects Q. The result of this intersection is the set $Q_T \equiv [\overline{T}_{min}, \overline{T}_{max}] \cap Q$.

Algorithm R: Find elements of Q which are within the target $\lfloor T_{min}, T_{max} \rfloor$.

1. Initialize the current subtree to be the root node of the data set Q: S $\leftarrow Q$
Initialize the return set Q_T to the empty set: $Q_T \leftarrow \{\,\}$

2. `Search(`S`)`: Search the subtree rooted at S:
{

2.1 If the data stored in S is contained by the target, then add S to Q_T:
`if` $\left(T^l_{min} \le x^l_S \le T^l_{max}\right)$ `then` $Q_T \leftarrow \{Q_T, S\}$

2.2 If the right subtree, S_R, exists and overlaps the target, (by eq. 1.19) then search S_R:
`if` $((S_R \neq 0) \wedge (b^i_{S,R} \ge T^i_{min}) \wedge (a^i_{S,R} \le T^i_{max}) \quad \forall (i \in \{1, \ldots, d\}))$ `then`
`search(`S_R`)`

2.3 If the left subtree, S_L, exists and overlaps the target, (by eq. 1.19) then search S_L:
`if` $((S_L \neq 0) \wedge (b^i_{S,L} \ge T^i_{min}) \wedge (a^i_{S,L} \le T^i_{max}) \quad \forall (i \in \{1, \ldots, d\}))$ `then`
`search(`S_L`)`

}

3. `Return(`Q_T`)`

Several aspects of this algorithm are worth emphasizing. First, elements of Q are only added to the return list, Q_T, in step **R**.2.1. The other steps, **R**.2.2 and **R**.2.3, are simply recursive calls back to **R**.`Search(`S`)`. Thus, step **R**.2.1 really does all the

"work" to build the return list. Secondly, the algorithm uses recursion to advance to deeper levels in the tree. Since the tree is balanced (by the partitioning strategy), one will always traverse to the full depth of the tree. Thus, the running time will be proportional to the depth of the ADT.

This observation makes it possible to estimate the complexity for the search operation. If N_{max} denotes the maximum number of objects that can be stored in an ADT of depth *D,* then the fact that the ADT is a binary tree implies:

$$\begin{aligned} N_{max} &= 2^D + 2^{D-1} + \ldots + 2^0 \\ &= (2(2^D) - 1) \end{aligned} \tag{1.20}$$

Solving for the tree depth, *D,* and substituting in the current number of objects, *N,* gives:

$$D = \left\lceil \log_2\left(\frac{N+1}{2}\right) \right\rceil \tag{1.21}$$

where the ceiling "⌈ ⌉" is used to indicate that non-integer results for *D* must be rounded up to the next integer.

Eq.1.21 implies that traversing the entire depth of the tree takes $O(\log N)$ operations, and since the running time of Algorithm **R** is proportional to the time required to traverse to the full depth, Algorithm **R** represents a substantial improvement over the linear performance of an exhaustive search.

As we noted, step **R**.2.1 actually assembles Q_T, and steps **R**.2.2 and **R**.2.3 simply setup recursive calls. When this algorithm is implemented on most compute hardware, this recursion incurs substantial call stack overhead. This is especially true since the "work" step, (**R**.2.1) is so short. CPU cycles dedicated to passing the argument list far outnumber those dedicated to doing useful work.

This observation suggests that it is prudent to implement a non-recursive version of Algorithm **R** which keeps a stack for remembering unfollowed branches of the tree. One such non-recursive implementation is shown in Algorithm **NR.**

In Algorithm **NR**, the monadic operators "++" and "--" are used to denote unit increments and decrements to `stackSize`. Note that this algorithm starts with an empty stack and pushes unfollowed right links which overlap the target onto the stack.

Algorithm NR: Find elements of Q which are within the target $[T_{min}, T_{max}]$ using a stack to avoid recursion.

1. Initialize the current subtree to be the root node of the data set Q: S ← Q
Initialize the return set Q_T to the empty set: $Q_T \leftarrow \{\}$
Set `stackSize = 0`

2. `Search(`S`)`: Search the current subtree rooted at S:
{

2.1 If the data stored in S is contained by the target, then add S to the return set:
`if` $\left(T^l_{min} \le x^l_S \le T^l_{max}\right)$ `then` $Q_T \leftarrow \{Q_T, S\}$

2.2 If the right subtree, S_R, exists and overlaps the target, push S_R onto the stack:
`if` $((S_R \neq 0) \wedge (b^l_{S,R} \ge T^l_{min}) \wedge (a^l_{S,R} \le T^l_{max}) \quad \forall (i \in \{1, \ldots, d\}))$ `then`
`PUSH(`S_R`):: stack(stackSize++)=` S_R

2.3 If the left subtree, S_L, exists and overlaps the target, set $S \leftarrow S_L$ and go to step 2:
`if` $((S_L \neq 0) \wedge (b^l_{S,L} \ge T^l_{min}) \wedge (a^l_{S,L} \le T^l_{max}) \quad \forall (i \in \{1, \ldots, d\}))$ `then`
$S \leftarrow S_L$
`goto 2`

2.4 If the stack is not empty, pop a new right link off the stack and go to step 2:
`if (stackSize ≠ 0) then`
`POP(`S`):: `S `= stack(stackSize);stackSize--`
`goto 2`
}

3. The stack must be empty so return the list: `Return(`Q_T`)`

Thus it first traverses all left links of a branch which overlap the target. When no more left links need to be checked, it pops the top right link off the stack and follows down the new branch in a left-links-first order.

Searching for Finite-Sized Objects

Thus far, the discussion has centered on construction of an ADT for point data in a hyperspace defined by the coordinate ranges of the data in the search set Q. However, the real utility of the ADT is for finding intersection candidates between finite-sized objects.

the extension of the discussion and algorithms in the preceding paragraphs to search sets of objects begins by recognizing that for two objects to intersect in a d-dimensional Euclidean space, the coordinate ranges of the objects must overlap in each of the d dimensions[71]. The Cartesian *bounding box* $[\bar{x}_{min}, \bar{x}_{max}]$ of a finite sized object in d dimensions is defined as the smallest Cartesian region which can contain the object. Figure 1-19 gives examples for planar and 3-D objects. This construction has the same form as the “target” region in the preceding paragraphs. Bounding boxes

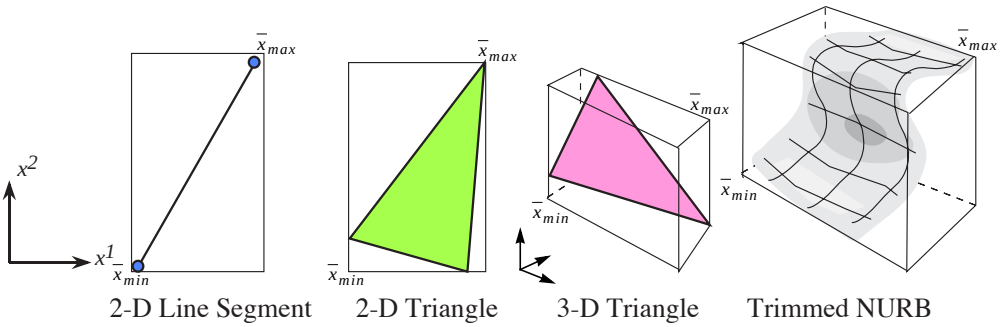


Figure 1-19: Cartesian "bounding boxes" of several finite geometric entities in 2- and 3-D.

may be defined for any finite geometric entity spanning any number of dimensions. It is precisely the abstraction of an object to its bounding box which permits the ADT to seamlessly consider heterogeneous collections of entities.

The observation that all pairs of coordinate ranges of two objects must overlap in order for them to intersect suggests a filtering for intersection checks. Rather than perform a detailed intersection computation of the objects in the set Q with the target region, we use the ADT only to return a list of those that *can* intersect. Specifically, the ADT returns a list of all objects in Q whose bounding boxes intersect that of the target.

This structure suggests the following simple test for checking the bounding box of the k^{th} object against the target region:

$$(x^i_{k,\,min} \le T^i_{max}) \wedge (x^i_{k,\,max} \ge T^i_{min}) \qquad \forall (i \in \{1, \ldots, d\}) \tag{1.22}$$

With this condition formalized, the ADT algorithm then introduces a clever mapping which reinterprets a d-dimensional bounding box $[\bar{x}_{min}, \bar{x}_{max}]$ as a single point in a $2d$-dimensional space.

$$[\bar{x}_{min}, \bar{x}_{max}] \rightarrow [x^1_{min}, x^2_{min}, \ldots, x^d_{min}, x^1_{max}, x^2_{max}, \ldots, x^d_{max}] \tag{1.23}$$

This mapping is applied to the bounding boxes of all the objects in the data set Q, which yields a point set in a hyperspace with $2d$ dimensions. With construction of this $2d$-dimensional hyperspace complete, the search space partitioning scheme from the preceding paragraphs is directly applicable.

The target region must also be mapped to the search space containing the point data. However, it must be mapped to a region so that it can *contain* the point data (as in the earlier example). This is accomplished by using the maximum and minimum

coordinate limits of the entire data set Q. Using the definition of the root bounding box $[\overline{a}_1\ \overline{b}_1]$ in eq. 1.15 yields (see also Figure 1-17):

$$\begin{aligned} \overline{T}_{min} &\rightarrow [a_1^1, a_1^2, \ldots, a_1^d, T_{min}^1, T_{min}^2, \ldots, T_{min}^d] \\ \overline{T}_{max} &\rightarrow [T_{max}^1, T_{max}^2, \ldots, T_{max}^d, b_1^1, b_1^2, \ldots, b_1^d] \end{aligned} \tag{1.24}$$

With this mapping of the target region in physical space into a region $[\overline{T}_{min}, \overline{T}_{max}]$ in the $2d$-hyperspace of the search, the containment criterion of eq. 1.23 becomes simply:

$$T_{min}^i \le x_k^i \le T_{max}^i \qquad \forall\, i \in \{1, \ldots, 2d\} \tag{1.25}$$

Eq.1.25 is precisely the containment criterion that was used in step 2.1 of Algorithms **R** and **NR**. However, since we are now searching for objects, rather than points, the levels of the ADT cycle through $2d$ dimensions. By representing the finite-sized, d-dimensional objects in set Q as points in $2d$ dimensions, the entire problem of finding intersection candidates for a list of objects has been reduced exactly to the problem of locating the points within a target region. Algorithms **R** and **NR** still hold, but now the return set Q_T contains the list of objects whose bounding boxes intersect that of the target. Since the objects are mapped to points in $2d$ dimensions, they are infinitesimally small in the search space, and therefore cannot cross the partitioning planes of the ADT. Thus each object may be unambiguously associated with a specific node of the ADT. The ADT used to store a set of d-dimensional objects is a binary tree in which each level corresponds to a partition in a $2d$ dimensional hyperspace.

Performance

Table 1.2 summarizes performance of an ADT implementation by the author. In this experiment a triangulation was constructed by computing the Delaunay triangulation of a randomized point set covering the region from the origin to (1, 1) in 2 dimensions. The corresponding ADT partitioned a 4-dimensional search space defined by the mapping in eq. 1.23. The experiment measured the look-up time for returning the lists of triangles whose bounding boxes intersected that of a randomly generated target region. Six triangulations were checked and the average look-up time for 100 targets is reported. Results using the ADT are compared with those of an exhaustive search over the triangulations in the column labeled “ratio”. Results are normalized by the average lookup time of the ADT on the smallest triangulation.

As expected, the tree consistently outperforms the exhaustive search, even for search sets with as few as 200 objects. With 500000 objects in the search set, the ADT was almost 100 times faster. Scatter in this data this data is most likely a result of the processor's use of Level 1 and Level 2 cache. No attempt was made to optimize the CPU's cache use for this example. Thus while some of the smaller triangulations certainly fit entirely into L1 or L2 cache, the larger triangulations required the CPU to fetch data off the memory chips, degrading performance slightly.

Table 1.2. Performance of ADT vs. Exhaustive search for finding intersection candidates with a Cartesian target in a 2-D triangulation, reported times are the average of 100 look-ups.

Number of Triangles, *N*	ADT	Exhaustive	Ratio
200	1	7.52	7.52
3200	2.06	16.76	8.23
11250	4.06	62.12	15.27
20000	9.53	120.53	12.65
80000	11.47	593.53	51.74
500000	37.65	3458.82	91.88

1.5.2 Painting Algorithms

A painting algorithm in computational geometry is one which traverses a graph marking or "painting" regions of cells which have some property in common. Figure 1-20 shows a sketch in which a set of triangles on an unstructured mesh are cut by a closed boundary. In this illustration, we wish to classify the triangles in the mesh as "inside" or "outside" of the boundary. This example assumes that the set of triangles that actually intersects the body is already known from some previous computation. Thus, all that remains is to efficiently classify the non-intersected triangles that remain.

Inside/Outside determination is an application of the *point-in-region* problem from computational geometry. For a body described with N geometric entities, an in/out check using an ADT can be designed which requires $O(\log N)$ operations per test. Therefore, with M triangles in the mesh, a naive approach to classifying the triangles would have a complexity of $O(M \log N)$. We seek an algorithm which will improve this performance to linear, or nearly linear time.

The approach begins by performing one full point-in-region query to set the status of any one of the unclassified triangles. This triangle is the "seed" for the painting algo-

Algorithm P: Paint all non-intersected triangles in the triangulation, *T*, as either "In" or "out".

```
1. for each triangle iTri {
2. if (itri.status ≠ unknown )then go to 1. (get next triangle)
3. Set current PaintColor ← Full_In_Out_check(iTri)
4. Paint(iTri):: Assigns the current PaintColor to iTri and passes this result to
                 unset neighbors.
   {
   4.a itri.status = PaintColor
   4.b Loop over face neighbors of iTri
       for each face neighbor iTri.tn {
          if (iTri.tn.status = unknown) then Paint(iTri.tn)
       }
   }
5. go to 1 }. (get next triangle)
```

Algorithms such as this have many uses in mesh generation and computational geometry as a whole. In addition to the in/out test illustrated here, they may also be used for a host of other partitioning schemes on unstructured meshes. More generally, such schemes may be generically used to traverse a graph propagating information from neighbor to neighbor along the logical links of the mesh.

The double teardrop geometry in Figure 1-21 consists of two closed surface triangulations which intersect tip-to-tail. Since they intersect, and are both closed polytopes, members of each triangulation lie within the other. If all the triangles of both bodies are placed in a single triangle list, a painting algorithm will completely classify the whole configuration with only four calls to the full in/out routine. Thus, the running time of the classification operation will be $O(M + 4\log M)$ which is nearly linear. The regions painted by each seed are indicated in the small sketch accompanying the geometry.

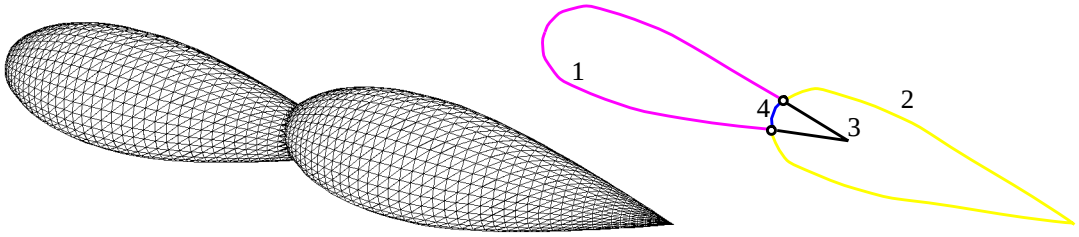


Figure 1-21: Surface triangulations for two intersecting teardrop shaped bodies. The labels on the sketch indicate regions painted by a single seed.

2. Topological Primitives, Intersection and Geometric Degeneracy

2.1 Introduction

This section focuses on the process of extracting the wetted surface of a configuration described by an arbitrary number of overlapping components. With this nominal goal in sight, the process of extracting this surface brings up many topics which are important not only to Cartesian mesh algorithms, but to both mesh generation as a whole and various aspects of computational geometry. The algorithms developed have general application to geometric problems. The final product of this section is a robust method for tessellating the exposed surface of a group of objects, and this algorithm has many applications not only in other disciplines of CFD, but in a variety of other fields within the computational sciences. The intersection problem introduces the fundamental topics of topological primitives, and geometric degeneracy, which is approached through the use of exact arithmetic and tie-breaking algorithms.

2.1.1 Motivation

A variety of successful Cartesian mesh schemes operate successfully without ever actually extracting the wetted surface of a configuration. The approaches of Melton[56] and Charlton[25], for example, retain internal portions of the geometry throughout the mesh generation and flow solution process. Such methods have produced several noteworthy results for a variety of complex configurations[58,57,3,25].

Still, removal of internal geometry offers several advantages worth considering. The foremost among these is that it greatly simplifies the mesh generation procedure by removing the ambiguities introduced by the possibility of having internal geometry. The sketch in Figure 2-1 gives an indication of some of the more obvious complications resulting from internal geometry.

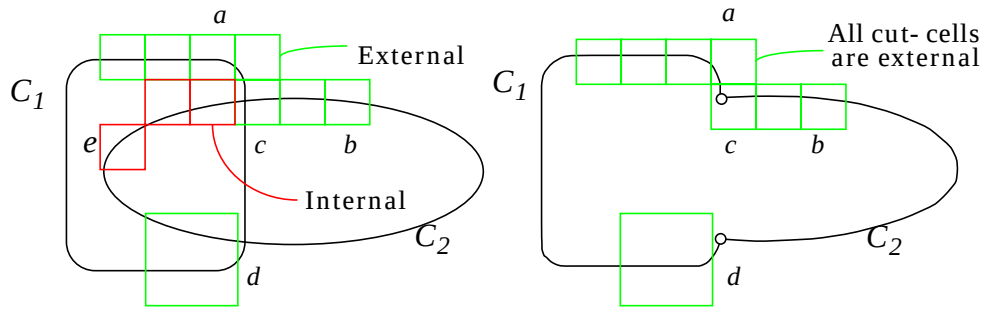


Figure 2-1: Component-based Cartesian mesh generation, with and without removal of internal geometry.

When internal geometry is present (as shown in the left of the sketch), body-cut Cartesian mesh cells may intersect a component boundary but be entirely encased within another component. Cell *e* in Figure 2-1 shows an example of such a cell. This cell must be tested for containment by C_1 before it can be rejected from the mesh. A second complication is that cells which do cut the outer surface, like *d*, may also intersect geometry which is completely contained. In this latter case, the boolean polygon subtraction $d = d - (C_1 \cup C_2)$ correctly computes the exposed region of *d*, but such operations can be computationally intensive. Finally, external cells, like *c*, may cut the geometry at the intersections between components. Since portions of the boundary of both components which *c* intersects make up the wetted surface of the configuration, it must be linked to surface fragments on each of its components. In addition, correctly computing cell *c*'s exposed region again requires a polygon subtraction similar to that for *d*.

Obviously, all of these cases can be considered when constructing the mesh generator without major effort, the more serious complication is less obvious. Since the strategy has allowed cells to intersect completely contained geometry, all intersected cells must be tested for this. Put another way, all intersections (internal or external) must be *classified*. Section 1.5.2 discussed the fact that point-in-polygon testing was at best an $O(\log N)$ operation. Thus by admitting internal geometry into the mesh generator, one has inherited an $O(\log N)$ test for every cut cell in the domain. When considering complex configurations, this is a potentially expensive proposition. If a configuration is described by $O(10^6)$ triangles, and the mesh has $O(10^7)$ cells, then around 10^6 of these would intersect the body. Since the mesh generation time can be linear in the number of cells, repeated classification of cut cells may swamp the procedure. In fact, a breakdown of the mesh generation time for various configurations performed by Refs. [56] and [3] have indicated that up to 60% of the processor time

was dedicated to resolving this issue. Communications with the authors of Ref.[25] have further substantiated these results.

While several strategies exist to reduce the time for intersection classification, a very attractive proposal is to avoid it completely. In the sketch at the right of Figure 2-1, the components have been intersected prior to mesh generation. Since no geometry exists within the wetted surface, any intersection with a Cartesian cell is, by design, exposed to the flow.

The Cartesian mesh generation strategies adopted by References [47] and [89] operate assuming that the geometry is described by either a single component or a collection of non-intersecting components. These methods use CAD packages to perform component intersection and extraction of the wetted surface. By intersecting the components as an integral part of mesh generation, we attempt to combine the flexibility and automation of a component-based approach, with the simplification of not having internal geometry.

Since the intersection is computed in the mesh generation phase, the complete geometry of all the components is available for later use. Thus the approach easily lends itself to automation and macroscopic control for parametric studies which move components or modify component geometry.

The component intersection algorithm developed here may be viewed as a general surface modeling technique. As such, it has many applications outside of Cartesian mesh schemes. The possibility of regridding the wetted surface as described by Löhner[54], makes it a useful tool for unstructured mesh generation. In addition, the automated block-structured (overset) method of Chan and Meakin[24] makes such an algorithm similarly useful for structured mesh applications.

Beyond CFD, the algorithm has application in many of the computational sciences. In computer science, ray tracing and solid modeling routines operate substantially faster if internal geometry is removed. In the field of nanotechnology, for example, microdevices are simulated by combining molecular models. Removal of the internal geometry substantially simplifies the descriptions of such structures and devices.

2.1.2 Important Topics

The design and construction of a geometric algorithm for intersecting components and extracting the wetted surface brings up many important topics in computational

geometry. Using this algorithm as a goal, this section introduces some of these topics and discusses the underlying problems and current research in these aspects of computational geometry. Since these topics are general, they arise in unstructured and structured mesh generation and surface modeling.

Topological Primitives: The specific tasks involved in assembling the complete intersection algorithm will be implemented through a set of predicates constructed through the use of topological primitives. A *topological primitive* is an operation which classifies a given input set into one of a constant set of possible outputs. A *binary* topological primitive has two possible outputs (*e.g.* true, false), while a *ternary* topological primitive has three. Such predicates can only classify existing data, they cannot construct new data. For example, they cannot provide the point of intersection between two line segments. They can only reveal if the two line segments do indeed intersect. In this way they provide information about the topology of their inputs. They may be used to establish a connectivity, trigger subsequent action, or describe a logical structure. We will discuss predicates for triangle-triangle and triangle-cube intersection, Delaunay triangulation, and point-in-polygon determination.

Geometric Degeneracy: When two objects are in *general* (or *simple*) position, they do not share common vertices, a common plane, or have improperly intersecting edges. Geometry is said to be in *special* position when some sort of degeneracy exists. Two line segments may share a common end point, or may improperly intersect where the endpoint of one lies exactly on the other segment. If more than two points are colinear, or more than three are coplanar or cocircular, the data is degenerate. *Arbitrarily* positioned data indicates that some objects in a set may be in special position. Although rare in nature, degenerate data is common (often by design) in analytic

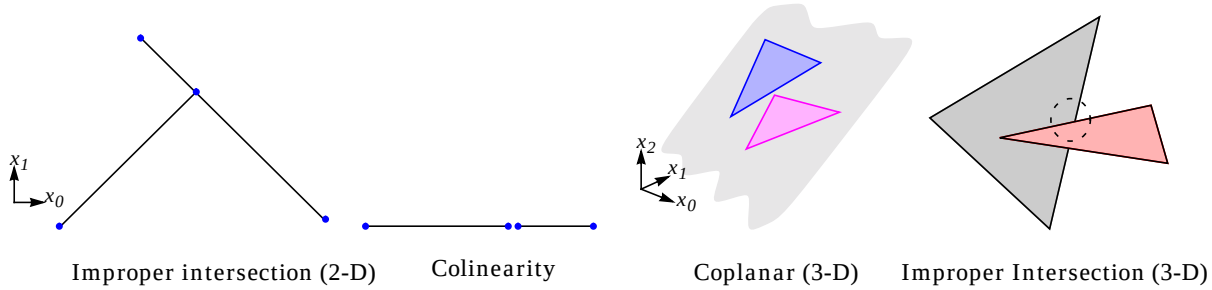


Figure 2-2: Some examples of geometric degeneracy in two and three dimensions.

geometry, and in the machine generated data output by CAD systems. Figure 2-2 illustrates a few common geometric degeneracies in two and three dimensions. The discussion will present an *algorithmic* approach for handling degenerate geometry, which fully avoids both tolerancing and handling of special cases.

Exact Arithmetic: Machine round-off error poses a formidable challenge to many geometric algorithms. Often geometric predicates are decided based on the sign of a result. While sign evaluation is exact for real numbers, the finite length, floating-point numbers used by compute hardware are not the set of reals. As a result, many topological primitives may find themselves trying to distinguish between numbers like 10^{-17} and exact zero. The algorithm must know if the result is round-off error, or significant. Common approaches to exact arithmetic are through the use of integer arithmetic[11,48] or via arbitrary precision floating point implementations[8,72,81].

2.2 Component Intersection

The problem of intersecting the various components of a given configuration and extracting the wetted surface may be viewed as a series of smaller. Although conceptually straightforward, efficient implementation of such an intersection algorithm is delicate. Each component is assumed to be described by its own surface triangulation. This is not a major restriction, since surface triangulations for individual components are generally easy to generate. The algorithm ultimately requires intersecting a number of non-convex polyhedra with arbitrary genus (an arbitrary number of “donut holes”). This generality makes convex polyhedra intersection algorithms inappropriate. Each intersected triangle must be broken up into smaller ones, which is a problem in constrained triangulation. Finally, the efficient deletion of the interior triangles requires inside/outside determination and neighbor painting.

2.2.1 Spatial Searches

Before beginning the actual intersection, we first construct an ADT as in section 1.5.1 and insert the triangles of all components into the tree. This provides a structure for returning the list of intersection candidates for any target triangle in $O(\log N)$ time.

As discussed earlier, the ADT is a hyperspace search technique which converts the problem of searching for finite sized objects in d dimensions to the simpler one of partitioning a space with $2d$ dimensions. Since searches are not conducted in physi-

cal space, however, objects which are physically close together are not necessarily close in the search space. This fact can degrade the tree's performance[77]. In an effort to improve lookup times, we therefore first apply a component bounding box filter on the triangles before inserting them into the tree.

Constructing this filter is a simple matter of checking all the triangles in the domain against the bounding boxes of the components. Since they cannot possibly participate in an intersection, triangles which are not contained by the bounding box of a component other than their own are not inserted into the tree.

The filtering process has two beneficial effects. First it reduces the tree size (depth) since fewer triangles are inserted. Thus subsequent look-ups will traverse a smaller tree. Obviously, this is a case dependent savings, but filtering typically removes from 25 to 75% of the triangles in a configuration from the ADT. Thus we create substantially smaller trees, which require less memory and are quicker to traverse.

The second advantage of the bounding box filter is that it increases the probability of encountering an intersection candidate in the tree. In other words, it improves the "hit rate" of the tree. The filter selectively removed only those triangles which could not possibly be involved in an intersection. Therefore, the tree is not crowded by irrelevant geometry.

2.2.2 Intersection of Generally Positioned Triangles in R^3

With the task of intersecting a particular triangle reduced to an intersection test between that triangle and those on the list of candidates provided by the ADT, the intersection problem is recast as a series of triangle-triangle intersection computations. Figure 2-3 shows a view of two intersecting triangles as a model for discussion. Each intersecting triangle-triangle pair contributes one *segment* to the final polyhedron that comprises the wetted surface of the configuration. At this point, it is sufficient to assume that the geometry is in general position. Thus, the intersections are always assumed to be non-degenerate. Triangles may not share vertices, and edges of triangle-trian-

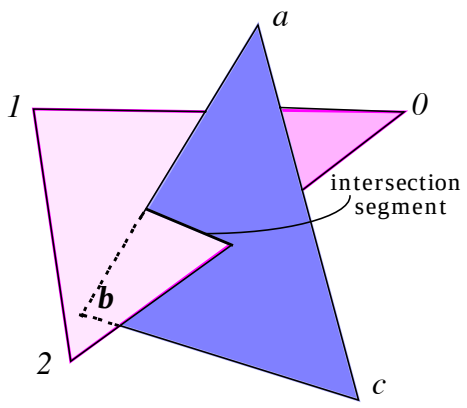


Figure 2-3: An intersecting pair of generally positioned triangles in three dimensions.

gle pairs do not intersect exactly. All intersections are proper. This restriction will be lifted in later sections with the introduction of an automatic tie-breaking algorithm.

Several approaches exist to compute such an intersection. Perhaps the most obvious, and certainly the most popular, is a simple slope-pierce test from analytic geometry. We can examine this approach using the sketch in Figure 2-4 which checks for the intersection of edge $\overline{ab}$ through triangle $\Delta_{0,1,2}$ from the previous figure. The slope-pierce test proceeds by first constructing p, the intersection of $\overline{ab}$ with the plane of $\Delta_{0,1,2}$. We then check to insure that p is between a and b ($a \ll p \ll b$) and finally if p is between a and b then it is checked for containment within $\Delta_{0,1,2}$ by verifying that the following vector cross-products all have the same sign.

$$\overrightarrow{02} \times \overrightarrow{p2} \qquad \overrightarrow{10} \times \overrightarrow{p0} \qquad \overrightarrow{21} \times \overrightarrow{p1} \tag{2.1}$$

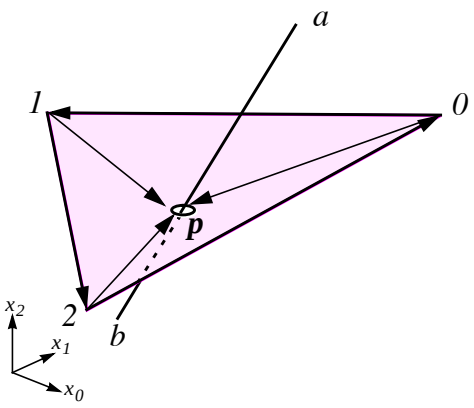


Figure 2-4: Sketch for a slope-pierce test of edge $\overline{ab}$ against triangle $\Delta_{0,1,2}$.

In analytic geometry, this test provides a reliable intersection check, and actually constructs the location of point p, (p_0, p_1 ,p_2) in the process. Computationally, however, the difficulties begin immediately. If one begins by expressing the line through (a_0, a_1, a_2) and (b_0, b_1, b_2) in standard form there is an obvious problem when this line is vertical (no-slope). Thus, vertical test edges already present a “special case”. While this difficulty can be circumvented through the use of a parametric representation of the line, a more fundamental and subtle difficulty still exists.

The computed intersection point, p, is a *constructed* entity. Operations which result in such new geometry may be referred to generically as “constructors”, and the geometry produced by constructors is necessarily known to different precision than the “given” data. While given data may be considered exact, constructed data has incurred round-off error during the construction process. Worse, since any slope-pierce test requires floating-point division, one has essentially lost control over the accuracy to which p is known. Consider IEEE 754 compliant, 32-bit hardware, single and double precision on these architectures uses 23 and 52 bits (respectively) to represent the fractional part of the normalized and rounded mantissa. Just as $10 \div 3$ results in a value not exactly expressible with a fixed number of base 10 digits, many real numbers cannot be exactly represented exactly in 23 or 52 bits. The division

operation needed for computing p may result in a loss of precision. Since its position cannot be trusted, if p falls near the edges or the vertices of the triangle, it would be indistinguishable from a point outside the triangle, and the implementation will fail.

A more fundamental observation about the slope-pierce test is to notice that it requires intermediate steps which generate new geometry, and this is not our goal. We wish to know if $\overline{ab}$ intersects $\Delta_{0,1,2}$. This is a question of topology, not of geometry.

A particularly attractive technique for determining if the segment intersects the triangle comes in the form of a Boolean test. This predicate can be performed robustly and quickly using only multiplication and addition. It therefore avoids the inaccuracy and robustness pitfalls associated with division of fixed width floating point numbers. It is useful to present a rather comprehensive treatment of this intersection primitive because subsequent sections on robustness will return to these relations.

For two triangles to properly intersect in three dimensional space, the following conditions must exist:

1. Two edges of one triangle must cross the plane of the other.
2. If condition (1) exists, there must be a total of two edges (of the six available) which intersect within the boundaries of the triangles.

A series of Boolean primitives may be constructed for these checks which have the attractive property that they permit one to establish the existence and connectivity of the segments without relying on the problematic computation of the pierce locations. If the locations of these points is needed at a later time, they may then be relegated to post-processing where they may be grouped together and, since the connectivity is already established, floating-point errors will not have fatal consequences.

The Boolean primitive for the 3-D intersection of an edge and a triangle uses the concept of the signed volume of a tetrahedron in 3-D. This signed volume is based on the well established relationship for the computation of the volume of a simplex, T, in d dimensions in determinate form (see for ex. [64]). The signed volume $V(T)$ of the simplex T with vertices $(v_0, v_1, v_2, ..., v_d)$ in d dimensions is:

$$d!V\left(T_{v_0 v_1 v_2 \ldots v_d}\right) = \det\begin{pmatrix} v_{0_0} & v_{0_1} & \cdots & v_{0_{d-1}} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{d_0} & v_{d_1} & \cdots & v_{d_{d-1}} & 1 \end{pmatrix} \tag{2.2}$$

where v_{k_j} denotes the j^{th} coordinate of the k^{th} vertex for $k \in \{0, 1, 2, \ldots, d\}$ and $j \in \{0, 1, 2, \ldots, d-1\}$. In three dimensions, eq. 2.2 gives six times the signed volume of the tetrahedron $T_{a,b,c,d}$.

$$6V(T_{a,b,c,d}) = \det\begin{pmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{pmatrix} = \det\begin{pmatrix} a_0 - d_0 & a_1 - d_1 & a_2 - d_2 \\ b_0 - d_0 & b_1 - d_1 & b_2 - d_2 \\ c_0 - d_0 & c_1 - d_1 & c_2 - d_2 \end{pmatrix} \tag{2.3}$$

This volume serves as the fundamental building block of the geometry routines. It is positive if the triangle $\Delta_{a,b,c}$ forms a counterclockwise circuit when viewed from an observation point located on the side of the plane defined by $\Delta_{a,b,c}$ which is opposite from d. Positive and negative volumes define the two states of the Boolean test while zero indicates that the four vertices are exactly coplanar. If the vertices are indeed coplanar, then the situation constitutes a "tie" which will be resolved with a general tie-breaking algorithm presented shortly (see §2.4). Of course, under the present assumption of generally positioned data, a tie can never occur. In applying this logical test to edge $\overline{ab}$ and triangle $\Delta_{0,1,2}$ in Figure 2-3, $\overline{ab}$ crosses the plane if and only if (iff) the signed volumes $T_{0,1,2,a}$ and $T_{0,1,2,b}$ have opposite signs. Figure 2-5 presents a graphical look at the application of this test.

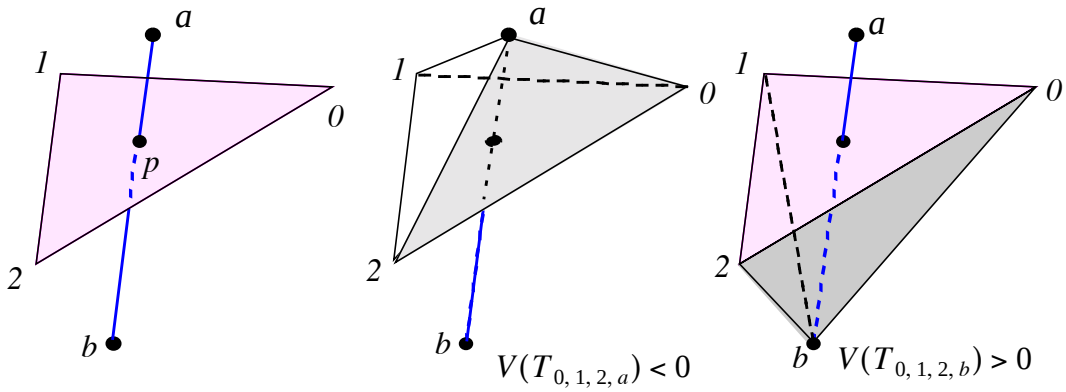


Figure 2-5: Boolean test to check if edge $\overline{ab}$ crosses the plane defined by triangle $\Delta_{0,1,2}$ through computation of signed volumes of two tetrahedra.

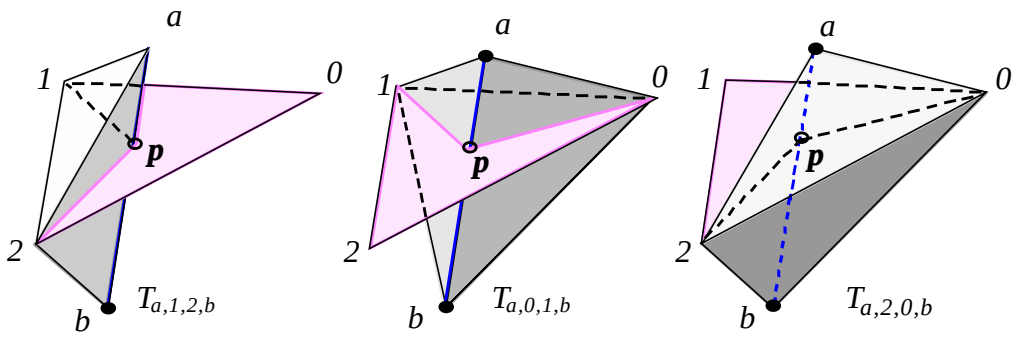


Figure 2-6: Boolean test for pierce of a line segment $\overline{ab}$ within the boundary of a $\Delta_{0,1,2}$.

With a and b established on opposite sides of the plane, all that remains is to determine if $\overline{ab}$ pierces within the boundary of the triangle. This is the case only if the three tetrahedra formed by connecting the end points of $\overline{ab}$ with the endpoints of the three edges of the triangle $\Delta_{0,1,2}$ all have the same sign, that is if:

$$\left[V(T_{a,1,2,b}) < 0 \wedge V(T_{a,0,1,b}) < 0 \wedge V(T_{a,2,0,b}) < 0 \right] \quad \text{or} \quad \left[V(T_{a,1,2,b}) > 0 \wedge V(T_{a,0,1,b}) > 0 \wedge V(T_{a,2,0,b}) > 0 \right] \tag{2.4}$$

is true. Figure 2-6 illustrates this test for the case where the three volumes are all positive.

Each triangle-triangle pair which intersects produces one line segment resulting from the intersection. It is the predicates in eqs.2.2 and 2.4 which determine the existence of all the segments resulting from intersections between triangle-triangle pairs. Within each intersected triangle, these segments may be connected to a linked list of all such segments that exist on that triangle. Thus the topology and connectivity of the segments within each triangle is known using only combinatorial operations. All that remains is to actually compute the locations of the pierce-points.

2.2.3 Construction of the Pierce Point Between an Edge and a Triangle

The slope-pierce test was rejected for determining if an edge intersected a triangle. The existence of a pierce-point between an edge and a triangle is a question of topology, making it appropriate to use a topological primitive to answer the question of

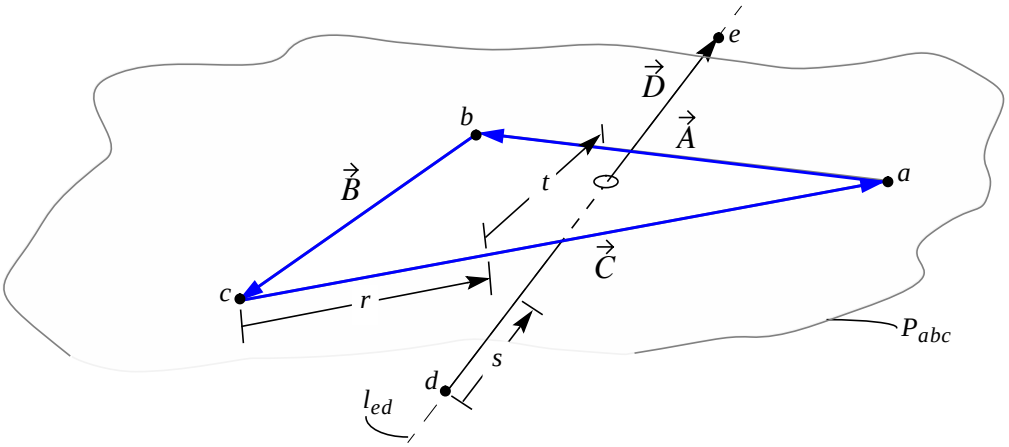


Figure 2-7: Parametric representation of a line defined by its endpoints and a plane defined by the three vertices of a triangle in 3-D.

existence. With the existence of the pierce-points established, and the connectivity associated with them already in hand, we may now design a constructor to generate the actual geometry of this pierce point.

Section 2.2.2 briefly noted that the direct use of slopes for constructing this location resulted in special cases when the edge has “no-slope”. The denominator of the equation of the line (in standard form) becomes zero and requires special treatment. A better alternative comes from expressing the line and the triangle with parametric representations.

Let $\vec{A} = \vec{b} - \vec{a}$, $\vec{B} = \vec{c} - \vec{b}$, $\vec{C} = \vec{a} - \vec{c}$, and $\vec{D} = \vec{e} - \vec{d}$ as indicated in Figure 2-7. The parametric representation of the plane of Δ_{abc} is $\vec{P}(r,t) = \vec{c} + r\vec{C} - t\vec{B}$, while the line, l_{ed} passing through points d and e is $\vec{l}(s) = \vec{d} + s\vec{D}$. The line and plane intersect when the values of the running variables r,s,t make $\vec{P}(r,t) = \vec{l}(s)$. Setting these two sets of parametric equations equal to each other constitutes a system of three equations in three unknowns.

$$\vec{c} + r\vec{C} - t\vec{B} = \vec{d} + s\vec{D} \tag{2.5}$$

Solving for s, r, and t yields:

$$\begin{aligned} s = \Big\{ & (c_0 - d_0)[(a_2 - c_2)(c_1 - b_1) - (a_1 - c_1)(c_2 - b_2)] \\ & -(c_1 - d_1)[(a_2 - c_2)(c_0 - b_0) - (a_0 - c_0)(c_2 - b_2)] \\ & +(c_2 - d_2)[(a_1 - c_1)(c_0 - b_0) - (a_0 - c_0)(c_1 - b_1)] \Big\} \frac{1}{\Gamma} \end{aligned} \tag{2.6}$$

and

$$
\begin{aligned}
r = \Big\{ & (e_0 - d_0)[(c_2 - b_2)(c_1 - d_1) - (c_1 - b_1)(c_2 - d_2)] \\
& -(e_1 - d_1)[(c_2 - b_2)(c_0 - d_0) - (c_0 - b_0)(c_2 - d_2)] \\
& +(e_2 - d_2)[(c_1 - b_1)(c_0 - d_0) - (c_0 - b_0)(c_1 - d_1)] \Big\} \frac{1}{\Gamma}
\end{aligned}
\tag{2.7}
$$

$$
\begin{aligned}
t = \Big\{ & (e_0 - d_0)[(a_2 - c_2)(c_1 - d_1) - (a_1 - c_1)(c_2 - d_2)] \\
& -(e_1 - d_1)[(a_2 - c_2)(c_1 - d_1) - (a_0 - c_0)(c_2 - d_2)] \\
& +(e_2 - d_2)[(a_1 - c_1)(c_0 - d_0) - (a_0 - c_0)(c_1 - d_1)] \Big\} \frac{1}{\Gamma}
\end{aligned}
$$

where the denominator Γ is:

$$
\begin{aligned}
\Gamma = \Big\{ & (e_0 - d_0)[(a_2 - c_2)(c_1 - b_1) - (a_1 - c_1)(c_2 - b_2)] \\
& -(e_1 - d_1)[(a_2 - c_2)(c_0 - b_0) - (a_0 - c_0)(c_2 - b_2)] \\
& +(e_2 - d_2)[(a_1 - c_1)(c_0 - d_0) - (a_0 - c_0)(c_1 - b_1)] \Big\}
\end{aligned}
\tag{2.8}
$$

Of course, it is possible that the denominator, Γ, may be zero in these equations. However, this occurs only when the line l_{ed} is parallel to the plane of the triangle. Since we only compute the pierce-points for segments that are already known to properly intersect the triangles, this case cannot arise. The topological checks have already eliminated it.

As a final comment, note that the parameterization for l_{ed} in eq. 2.5 is such that $\vec{l}(s)$ returns a point on the segment $\overline{ed}$ for $s \in [0, 1]$. Similar bounds apply for r and t. Thus we can check the consistency of the geometry constructed by eqs. 2.6-2.8.

2.2.4 Retriangulation of Intersected Triangles

The final result of the intersection step is a list of segments linked to each intersected triangle, and the locations of the pierce-points for the edges which intersect other triangles in the configuration. The complete list of all the segments attached to all the intersected triangles constitutes the geometric intersection of the polyhedra describing the components within the geometry. Figure 2-8 contains an example showing what has been constructed thus far. On the left, the figure shows a model

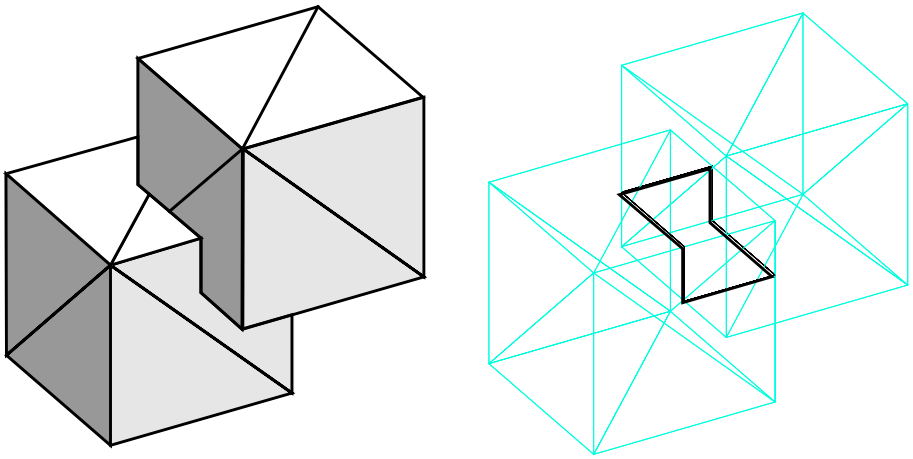


Figure 2-8: Left: Example configuration of two intersecting cubes. Right: wireframe showing intersection with segments linked to triangles.

configuration with two intersecting cubes. On the right, the cubes are shown in wireframe with the segments that make up the intersections highlighted.

Figure 2-9 examines the situation on any one of the intersected triangles. The figure shows a model of a generic intersected triangle linked to 3 segments. The segments divide the intersected triangle into polygonal regions which are either completely inside or outside of the body. In order to remove the portions of these triangles which are inside, we triangulate these polygonal regions within each intersected triangle and then reject the triangles which lie completely inside the body.

In the sketch, the segments resulting from the intersection calculation are highlighted. These segments serve as the constraints in the triangulation which decomposes the large triangle Δ_{abc}. Since the segments may cut the triangle arbitrarily, a pre-disposition exists for creating triangles with arbitrarily small angles. In an effort to keep the triangulations as well behaved as possible, we employ a two dimensional Delaunay triangulation algorithm within each original intersected triangle. Using

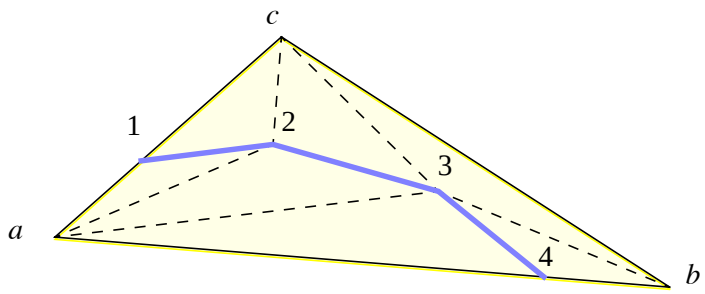


Figure 2-9: Decomposition of intersected triangle using a constrained Delaunay triangulation algorithm (constraining segments shown as heavy solid lines).

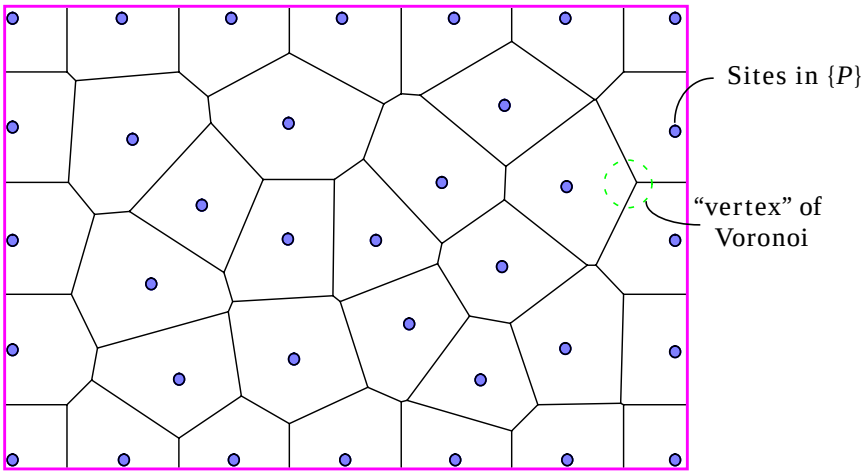


Figure 2-10: Voronoi diagram $V(P)$ of a set $\{P\}$ with 35 sites

the Delaunay triangulation not only guarantees that the triangulation will exist, but it also maximizes the minimum angle in the triangulation. Using the intersection segments as constraints, the algorithm runs within each intersected triangle producing new triangles which may be uniquely characterized as either completely inside or outside.

Voronoi Diagrams

Before turning directly to the Delaunay triangulation, it is useful to first examine the *Voronoi diagram* of a set of sites in a plane. The Voronoi diagram is an extremely important structure in computational geometry, in fact it has been called “a geometric structure second in importance only to the convex hull”[64]. Voronoi diagrams have been encountered in a variety of disciplines and have been applied to problems ranging from the geometry of soap bubbles and crystal growth patterns to the locations of fire towers[78]. The Voronoi diagram is also known as the Dirichlet tessellation since it was first discussed by Dirichlet in 1850. A history of the subject is presented in reference [7].

For a collection of sites in the plane $P = \{P_1, P_2, \ldots, P_n\}$, the Voronoi diagram $V(P)$ of the set is the planar straight-line graph constructed by drawing the line segments which separate the plane into regions that are closer to any particular member of P than to any other site in the set. Figure 2-10 shows an example for a set with 35 sites. Each of the regions surrounding the sites identifies locations in the plane which are closer to that site than to any other site in the set. The analogy of soap bubbles is clear, if the nucleation sites for the bubbles are those in P, and the bubbles

are inflated at a constant rate, $V(P)$ is a graph of the resulting structure when the bubbles can inflate no more.

Some properties associated with the Voronoi diagram include:[1]

1. The Voronoi region associated with any site, $V(P_i)$ is convex.
2. $V(P_i)$ is unbounded iff P_i is on the convex hull of the set of sites, *P*.
3. The "vertex" of a Voronoi diagram is the point at which 3 or more Dirichlet Regions meet (see Figure 2-10). If *v* is the vertex at the junction between $V(P_1)$, $V(P_2)$ and $V(P_3)$, then *v* is the center of the circle $C(v)$ which passes through P_1, P_2, and P_3.
4. $C(v)$ is the circumcircle for the Delaunay triangle corresponding to *v*.
5. The interior of $C(v)$ is site free.
6. If P_j is the nearest neighbor to P_i then $\overline{P_iP_j}$ is an edge of the Delaunay triangulation of *P*.
7. If a circle exists through P_i and P_j, which contains no sites then $\overline{P_iP_j}$ is an edge of the Delaunay triangulation of *P*.
8. $V(P_i)$ is unique.

The duality of the Voronoi diagram and Delaunay triangulation implies that many of these properties correspond to properties of the Delaunay triangulation as we will see below.

Delaunay Triangulations

The *Delaunay triangulation* $D(P)$ of the set of sites, *P*, is the straight-line mesh dual of the Voronoi diagram $V(P)$, provided that no four sites are cocircular. Figure 2-11 shows the Delaunay triangulation for the sites used to demonstrate the Voronoi diagram (Figure 2-10). Delaunay showed that this dual will always be a triangulation as long as no four sites in *P* are cocircular[33]. There is a one-to-one correspondence

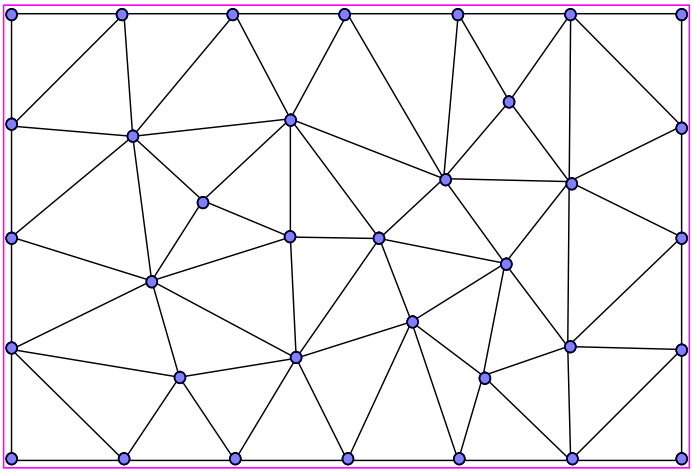


Figure 2-11: Delaunay triangulation of a set of 35 sites (see Voronoi diagram in Figure 2-10

1. Following from reference [64] and others.

between edges in the Delaunay triangulation and edges in the Voronoi. Examination of the examples in Figures 2-10 and 2-11 reveals that this correspondence does not necessarily mean that the dual segment in $D(P)$ actually crosses its corresponding edge in $V(P)$.

Some properties of Delaunay triangulation include:[1]

1. $D(P)$ is the straight-line dual of $V(P)$.
2. Each triangle in $D(P)$ corresponds to a vertex in $V(P)$.
3. Each edge in $D(P)$ corresponds to an edge in $V(P)$.
4. Each node in $D(P)$ corresponds to a region in $V(P)$.
5. The boundary of $D(P)$ is the convex hull of the sites. (*cf.* Voronoi prop. 2.)
6. The circumcircle for a triangle Δ_{P_1, P_2, P_3} in $D(P)$ is centered at Voronoi vertex *v*, at the junction between $V(P_1)$, $V(P_2)$ and $V(P_3)$. (*cf.* Voronoi props. 3-4)
7. The circumcircle for a triangle Δ_{P_1, P_2, P_3} in $D(P)$ contains no other site in *P*. (*cf.* Voronoi prop. 5.)
8. There exists an edge-circle which passes through the two sites of every edge in $D(P)$ which is site free. (*cf.* Voronoi prop. 7.)
9. The Delaunay triangulation maximizes the minimum angle of the triangulation of *P*.
10. $D(P)$ is unique and always exists.

Properties 9 and 10 make the Delaunay triangulation a particularly attractive method for retriangulating the intersected triangles like that shown in Figure 2-9. Since the components will intersect arbitrarily, these triangles may be very complicated, and "glancing" intersections introduce the likelihood of having triangles with very small angles. Property 9 implies that we will maximize (as much as possible) the small angles within the final triangulation. Property 10 guarantees that the approach can never fail.

Delaunay Triangulation by Successive Point Insertion

A variety of approaches exist to construct the Delaunay triangulation of a point set (see surveys in Refs.[9] and [55]). However, since each triangulation to be constructed starts with the three vertices of the original intersected triangle (vertices *a*,*b*,*c* in Fig. 2-9), the incremental algorithm of Green and Sibson[44] is particularly appealing. Starting with the three vertices defining the original triangle, the pierce-points associated with the segments are successively inserted into the evolving triangulation. After all the pierce-points are inserted, the constraints are enforced.

1. This is a partial list, see references [71], [55], and [9] for more.

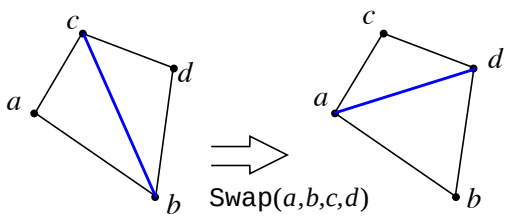


Figure 2-12: Demonstration of the Swap(a,b,c,d) operation which exchanges the diagonal of the quadrilateral ($cb \rightarrow ad$)

The Green and Sibson algorithm is incremental, because it adds the sites one at a time, while recovering a Delaunay triangulation after each site insertion. The algorithm makes extensive use of "edge swapping" as shown in Figure 2-12. After identifying an edge to be swapped, (edge $\overline{bc}$ in Fig.2-12), one identifies the quadrilateral formed by the two triangles that share the edge. Swap(a,b,c,d) then reconfigures this quadrilateral by exchanging the diagonal. The operation is completely local to the quadrilateral, and thus has no effect on triangles in the rest of the mesh.

Figure 2-13 provides an overview of the Green and Sibson incremental triangulation algorithm[44]. The method begins with a pre-existing Delaunay triangulation. In the present case, this is simply the intersected triangle which is to be retriangulated. In general, one begins by creating a triangle which encases all the sites in the set P.

Step 1. Insert the next site, p, into the triangulation, and locate the triangle $\Delta_{a,b,c}$ which contains this site. Under the assumption of general position, this site can never fall directly on an edge in the mesh, there must be a triangle which contains it.

Step 2. Connect the new site with vertices of the triangle which contains it by adding the three edges $\overline{pa}$, $\overline{pb}$, and $\overline{pc}$. These new edges are all incident upon p, and since the original triangulation was Delaunay, p must be a nearest neighbor of a, b, and c. Thus, the three new edges are automatically Delaunay. The original edges of the triangle, however, are not. Therefore, the original edges $\overline{ac}$, $\overline{ba}$, and $\overline{cb}$ now become "suspect" edges.

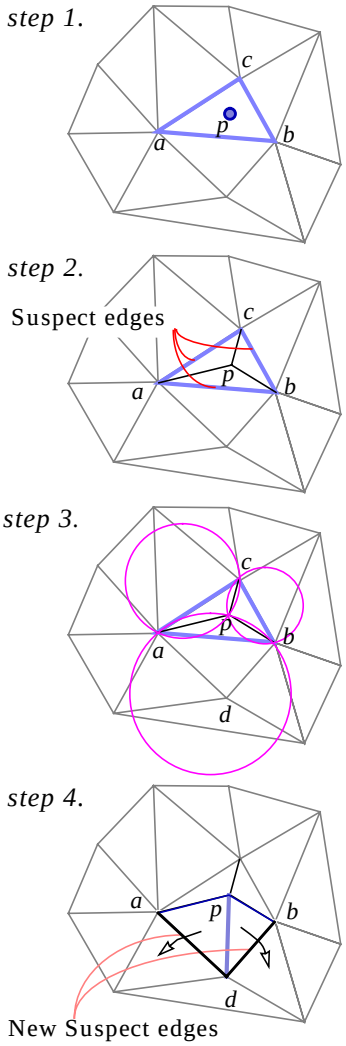


Figure 2-13: Illustration of Green and Sibson Delaunay triangulation algorithm.

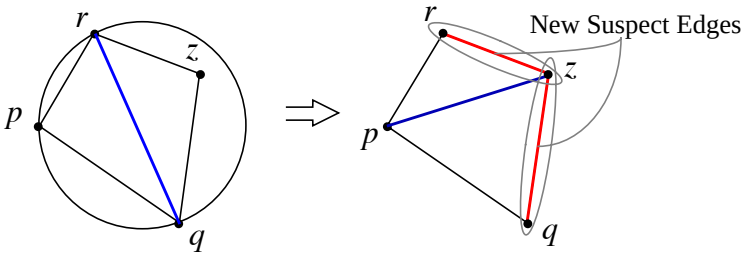


Figure 2-14: Incircle testing of point *z* for containment within the circumcircle of (*p*,*q*,*r*). Since *z* is contained, the diagonal of the quadrilateral *pqzr* is swapped (*qr*→*pz*).

Step 3. Apply the *incircle* predicate to the suspect edges. The incircle predicate is based on the circumcircle property (prop. 7, page 53) which states that the circle through the three vertices of a triangle in a Delaunay triangulation contains no other sites. The predicate is applied to each of the suspect edges. Each suspect edge constructs the quadrilateral formed by its two neighboring triangles. It then constructs a circle through the new site, *p*, and tests the fourth point of the quadrilateral for containment. If the test point falls within the circle, the diagonal of the quad is swapped (see Fig.2-12). Figure 2-14 shows an example in which the test point (*z*) fails, resulting in a swap.

Step 4. Swap and propagate. In Figure 2-13, the original suspect edges are $\overline{ac}$, $\overline{ba}$, and $\overline{cb}$. Of these, only $\overline{ab}$ needs to be swapped. When an edge is swapped, new suspect edges are identified on the boundary of the quadrilateral. Of these, only those not incident upon point *p* require testing, since we already know from *step 2* that edges incident upon *p* are Delaunay. Thus the incircle test gets propagated *forward* through the mesh. In Figure 2-13, the new suspect edges are $\overline{ad}$, and $\overline{db}$, and the small arrows indicate the direction of propagation. In the demonstration of the incircle predicate shown in Figure 2-14, forward propagation dictates that edges $\overline{qz}$ and $\overline{zr}$ become the new suspects. Each new suspect edges recursively calls the incircle predicate, swapping edges if necessary. The edge swapping continues until all suspect edges result in no new swaps. When the edge swapping terminates, the new mesh is Delaunay, and the algorithm returns to *Step 1* to insert the next site. The process repeats until all sites in the set {*P*} are inserted.

Running time for the Green and Sibson algorithm is strongly dependent upon the method used to locate the triangle which contains a new site, and the order in which the sites are added. For *N* sites, each proximity query in *step 1* may be conducted in

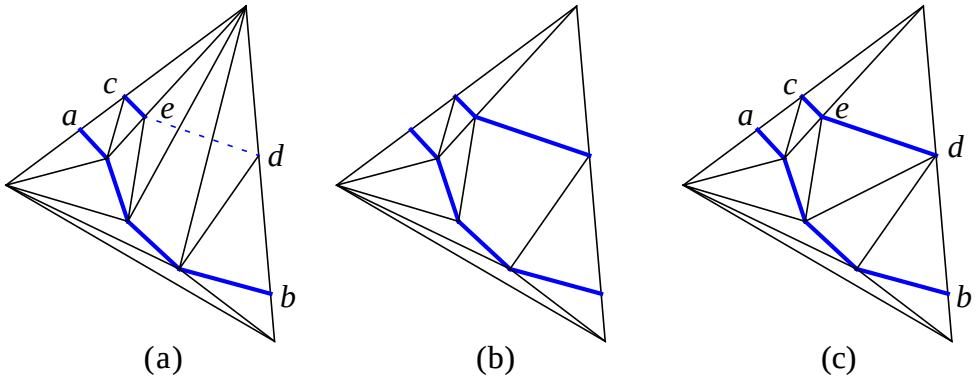


Figure 2-15: Enforcement of constraining segments through edge deletion and retriangulation.

$O(\log N)$ time if one pre-sorts the sites, or $O(N^{1/2})$ using a random site choice combined with a stencil-walk[60]. If sites are inserted in an order which minimizes swapping, then the algorithm has a best case complexity of $O(N \log N)$. The upper complexity bound assumes that a particularly poor site insertion strategy may produce $O(N^2)$ swapping operations after each site is inserted. Inserting sites in a random order has a high probability of avoiding the pathological quadratic upper bound[46,9].

Constrained Delaunay Triangulation

At this juncture, the intersected triangles have been decomposed into smaller triangles, using the pierce-points as sites in the Delaunay algorithm. What remains is to insure that the segments resulting from the intersection of each triangle-triangle pair are edges in the triangulation. The segments need to be constraints of the triangulation on each intersected (original) triangle (as shown in Figure 2-9, page 50).

A *constrained Delaunay triangulation* is one which contains a set of prescribed edges such that the circumcircle of each triangle contains no other vertex of the mesh which is *visible* to it[27,82]. Two vertices are visible two each other if a straight line joining them does not intersect a constraint.

Each constraining segment is first checked to see if it is already in the set of edges of the triangle. If it is not, then the edges it crosses are recursively swapped until the edge becomes an edge in the triangulation. Figure 2-15 shows a demonstration of this process. Initially (a) all the segments joining *a* and b are already edges in the triangulation. However, constraining edge $\overline{ed}$ is not. In (b), the edges that $\overline{ed}$ intersects

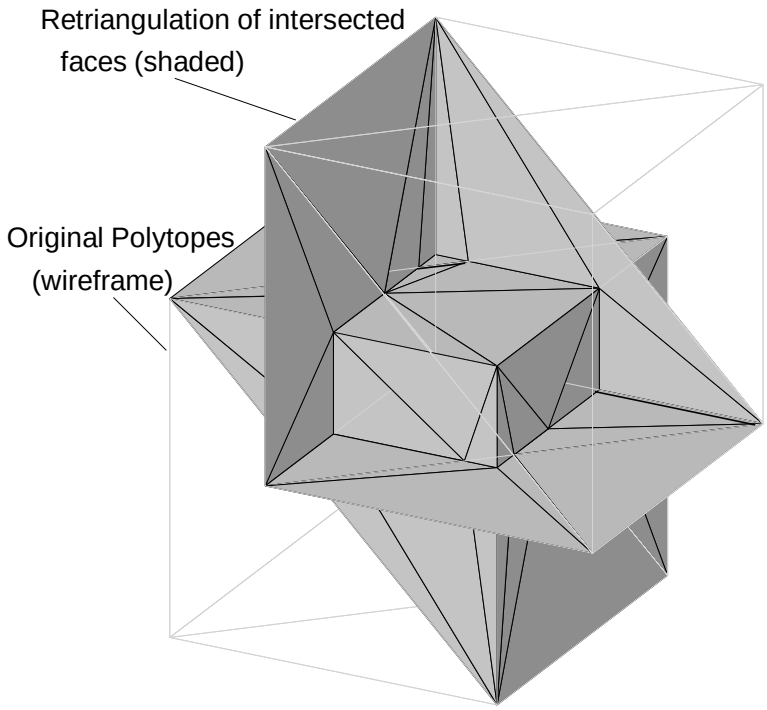


Figure 2-16: Decomposition of intersected triangles into triangles which lie completely inside or outside the configuration.

are removed and *ed* is inserted. The vacant polygons on either side of *ed* are then retriangulated in accordance with the incircle test (see [55,27,82] for further detail).

After enforcement of the constraints, each intersected triangle has been decomposed into small triangles which lie entirely inside or outside of the wetted surface of the triangulation. Figure 2-16 presents a view of the situation in 3-D, using the "two cubes" configuration from Figure 2-8 (page 50). The figure shows that after the intersection, no triangle is properly intersected by any other.

2.2.5 The Incircle Predicate

The incircle predicate is the cornerstone of the Delaunay algorithm in §2.2.4. Figure 2-14 on page 55 presented this predicate graphically. Now we wish to formalize the construction of this test, using the topological primitives designed for the intersection computation. Relating the incircle test to the signed volume calculation of eq. 2.3 starts by recognizing that if one projects the 2-D coordinates (x,y) of each point in the incircle test onto a unit paraboloid $z = x^2 + y^2$ with the mapping:

$$(k_x, k_y) \rightarrow (k'_x, k'_y, k'_z), \text{ where } k'_x = k_x, \quad k'_y = k_y, \text{ and } k'_z = k_x^2 + k_y^2 \qquad (2.9)$$

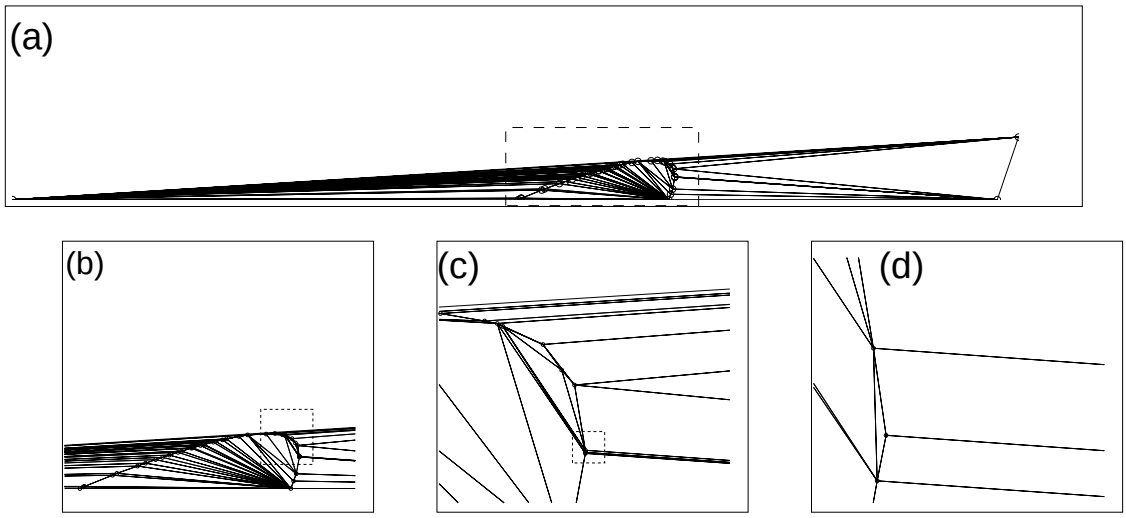


Figure 2-17: Retriangulation within a large fuselage triangle pierced by a wing leading edge component with significantly higher resolution. The 52 segments describing the intersection of the leading edge constrain the triangulation.

Using this mapping, the four points of the quadrilateral in the 2-D incircle predicate project to form a tetrahedron in 3-D. Therefore, the incircle predicate may be viewed precisely as an evaluation of the volume of a tetrahedron in the mapped coordinates[1]. As a result, `InCircle(`p,q,r,z`)` becomes precisely an evaluation of $V(T_{p'q'r'z'})$ using eq. 2.3. When $V(T_{p'q'r'z'}) > 0$, then z lies within the circle defined by p, q, and r and edge $\overline{qr}$ must be swapped to $\overline{pz}$ for the triangulation to be Delaunay (see Figure 2-14 on page 55). This mapping properly casts the incircle predicate as a topological primitive which is desirable for the same reasons discussed in §2.2.2 concerning the intersection of triangles.

Figure 2-17 shows an example of the retriangulation procedure applied within a single large fuselage triangle that has been intersected by a wing leading edge. Since the wing leading edge has much higher resolution requirements, its triangulation is substantially more refined. In all, 52 segments from the wing leading edge constrain the triangulation. This example is interesting because it demonstrates the need for robustness within the intersection and retriangulation algorithms. The length scales created by the intersection differ in magnitude by approximately 2^9 (see footnote[2]). Component data is considered “exact” in single precision, and the intersection points

1. When the mapping in eq. 2.9 is applied to set $\{P\} \rightarrow \{P'\}$, $D(P)$ maps to the lower convex hull of $\{P'\}$. See [64] or [50] for details.
2. It is useful to think in base 2 here, because it reveals that the first 9 bits of the numbers are identical (assuming the exponent on all the data is the same - as it will be if the geometry is not too near the origin). In double precision on a 32-bit machine, this implies that 41 bits are still left to uniquely characterize the geometry - the intersections can get substantially “worse” before one runs out of resolution.

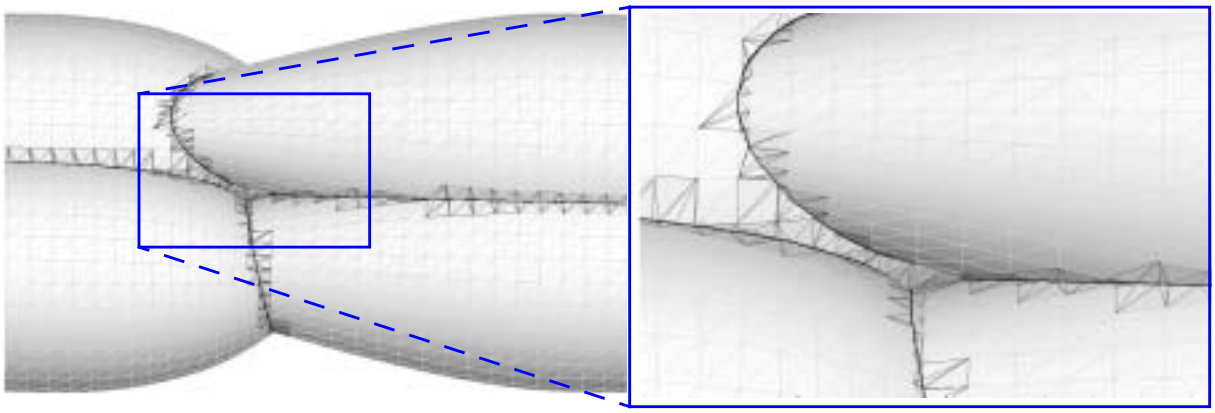


Figure 2-18: Example of constrained Delaunay retriangulation of intersected triangles for four intersecting teardrop bodies.

are computed from eqs.2.6-2.8 using double precision. This example involved no tie-breaking. However, the succession of embedded enlargements in Figure 2-17 emphasizes the degree of irregularity possible in the resulting triangulations, and it underscores the demand for a robust implementation of the fundamental geometry routines.

Figure 2-18 presents a 3-D example with four intersecting teardrop shaped bodies. The constrained Delaunay triangulation algorithm ran within all in the intersected triangles, retriangulating, and constraining each triangulation with the segments from the intersection. In the figure, the constrained Delaunay triangulations are shown with heavy lines, and the arbitrary nature of the intersections is apparent.

2.2.6 Inside/Outside Determination

The intersection and constrained triangulation routines of sections 2.2.2 and 2.2.4 have resulted in a set of triangles which may now be uniquely classified as either internal or exposed to the wetted surface of the configuration. The only step left is then to delete the internal triangles. This is a specific application of the classic “point-in-polyhedron” problem from computational geometry. We approach this problem with a ray-casting approach. This method fits particularly well within the framework provided by the proximity testing algorithm and topological primitives presented earlier (§1.5.1 and §2.2.2).

The point-in-polyhedron problem is a generalization of the point-in-polygon problem in the plane. Figure 2-19 illustrates the two common approaches to this problem by testing q for containment in a polygon P. On the left side of the sketch, the *winding number*[39] is computed by completely traversing the closed boundary P from the per-

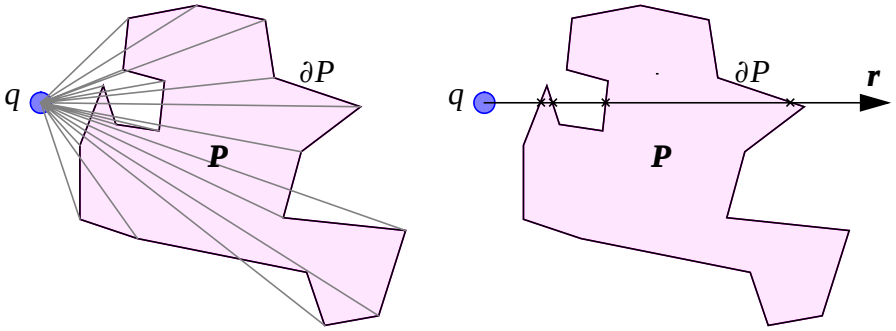


Figure 2-19: Illustration of point-in-polygon testing using the (left) winding number and (right) "ray-casting" approaches

spective of an observer located at q, and keeping a running total of the signed angles between successive polygonal faces. As shown in the left of the sketch, if $q \notin P$ then the positive angles are erased by the negative contributions, and the total angular turn is identically zero. If, however, $q \in P$, then the winding number is 2π. For a simplicial polyhedron in 3-D, we can sum the solid angle included by each triangle in ∂P and use a painting algorithm (§1.5.2) to traverse the faces of the polyhedron. In three dimensions, of course, $q \in P$ will be indicated by a solid angle of 4π.

The alternative to computing the winding number is to use a *ray-casting* approach. As indicated in the right sketch of Figure 2-19, one casts a ray, r, from q and simply counts the number of intersections of r with ∂P. If the point lies outside, $q \notin P$, this number is even, if the point is contained, $q \in P$, this result is odd.

While both approaches are conceptually straightforward, they are considerably different computationally. Computation of the winding number involves floating-point computation of many small angles, each of which is prone to round-off error. The running sum makes these errors cumulative, increasing the likelihood of robustness pitfalls. In addition the method poses the topological question "Inside or outside?" with a floating-point comparison to zero or 2π. Ray-casting poses the inside/outside question in topological (Does it cross?) terms.

A second drawback of using the winding number comes from consideration of improving its asymptotic performance. For a polygon with N faces, both tests may be conducted in linear time. However, the accumulation in the winding number necessitates a visit to each face, making it unlikely to easily improve upon this performance. Ray-casting, on the other hand, may be easier to accelerate, since the facets can be pre-sorted. For example, if the ray is always cast in $+x$, then the polygonal segments

of ∂P could be sorted in y, and inserted into a binary tree. The list of intersection candidates for the ray could then be identified with $O(\log N)$ operations.

Obviously the ray-casting approach fits well with the search and intersection framework that we developed earlier. The preceding sections demonstrated that both the intersection and triangulation algorithms could be based upon Boolean operations checking the sign of the determinant in eq. 2.3, and the same is true for the ray casting step. Locate q at the centroid of any triangle in the configuration. Then assume r is cast along a coordinate axis ($+x$ for example) and truncated just outside the $+x$ face of the bounding-box for the entire configuration. This ray may then be represented by a line segment from the test point (q_0, q_1, q_2) to $(\lceil \partial\Omega \rceil_x + \varepsilon, q_1, q_2)$ and the problem reduces to a proximity query of §1.5.1 followed by the segment-triangle intersection algorithm of §2.2.2. The tree returns the list of intersection candidates while the signed volume in eq. 2.3 checks for intersections. Counting the number of such intersections determines a triangle's status as inside or outside. The only caveat in this approach is that all the triangles in the configuration must be present in the ADT, since we don't know *a-priori* where the rays will pierce.

The painting algorithm presented in section 1.5.2 offers a method to avoid casting as many rays as there are triangles. This allows each tested triangle to pass on its status to the three triangles which share its edges. The algorithm then recurses upon the neighboring triangles until a constrained edge is encountered at which time it stops. In this way the entire configuration may be classified with few ray casts.

Figure 2-20 illustrates the ray-casting and painting procedure. The figure shows a configuration of 3 teardrop shaped bodies, similar to those used to demonstrate the

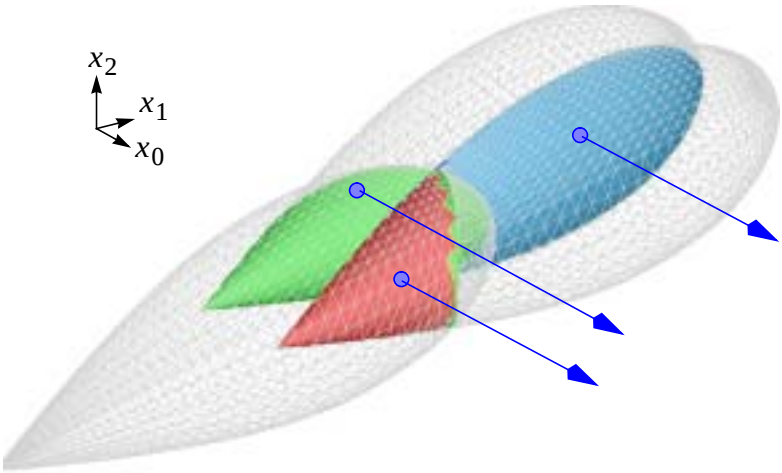


Figure 2-20: Example of ray-casting combined with mesh painting. The three internal regions shown were identified and painted using one ray per region

retriangulation procedure. The component triangulation is shown in wireframe, and three internal regions are shown painted. Each of the internal regions was seeded with a single ray cast, and filled in with the painting algorithm. All the original triangles on this entire configuration were painted with a total of 12 rays, corresponding to the 12 regions of the geometry.

Figure 2-21 and Figure 2-22 present two brief examples. Figure 2-21 is an example of a helicopter comprised of 82 components with 320,000 triangles. The configuration includes external stores and armaments. The complete intersection, retriangulation and removal of interior geometry required ~200sec on workstation with a 195 Mhz MIPS R10000 processor. Figure 2-22 shows two close-ups of the inboard nacelle on a high-wing transport configuration. The frame on the left shows the final geometry after intersection, retriangulation and trimming, while the right frame shows a view inside by removing the outboard section of the wing with a cutting plane through the center of the nacelle. This configuration consisted of 86 components described by 214,000 triangles.

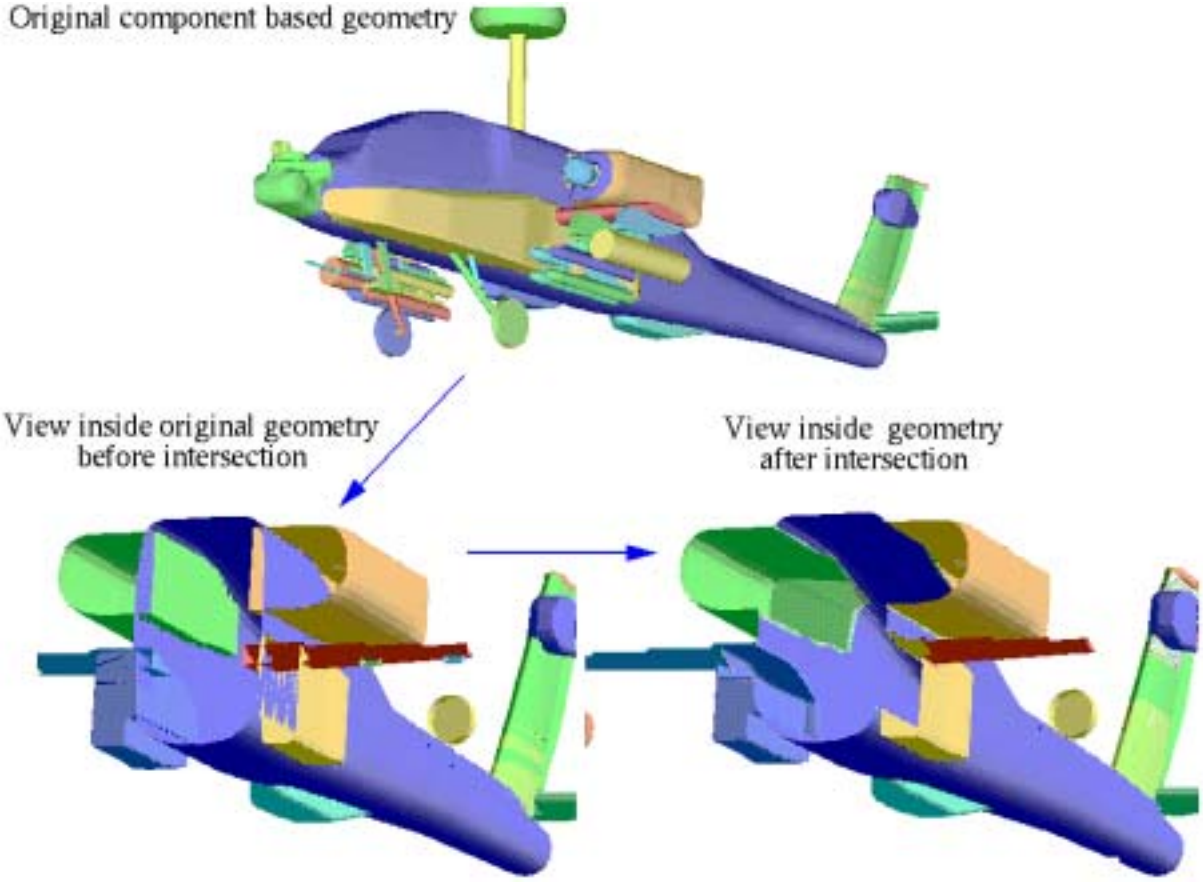


Figure 2-21: Helicopter example containing 82 components including external stores and armaments.

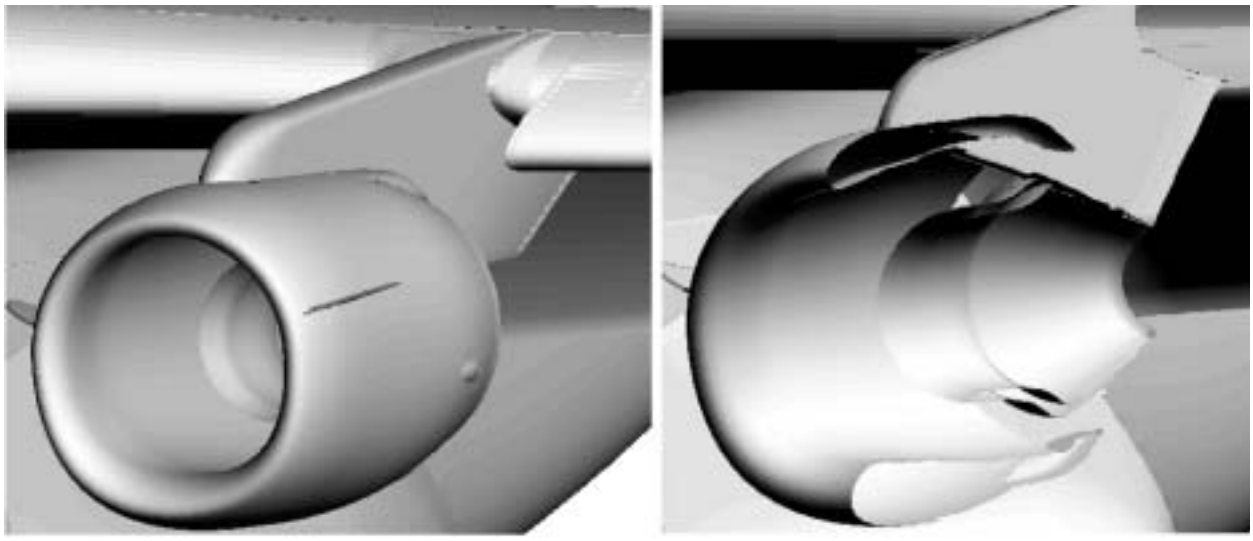


Figure 2-22: (left) Close-up of inboard nacelle of high-wing transport after intersection, retriangulation and removal of interior geometry. The frame shows leading edge slat, wing section, pylon, nacelle, nacelle strakes, and various other engine components. (right) View inside after removal of internal geometry. The configuration consisted of 86 components described using 214,000 triangles

The HWT geometry in Figure 2-22 is a more detailed version of that shown earlier in Figure 1-16 on page 24. There were over 10400 intersected triangles in this configuration, and the average intersected triangle was divided into 6 smaller ones by the constrained delaunay procedure. The largest of these retriangulation calls divided an intersected triangle into 232 smaller triangles. During the intersection, on average the ADT returned 8 intersection candidates for each test triangle, and at most 1131 such candidates were returned.

2.3 Floating-Point Filtering and Exact Arithmetic

With the intersection, retriangulation and ray casting algorithms all wholly dependent upon the determinant computation of eq. 2.3, it is imperative to insure accurate evaluation of this volume. In fact, all of the operations establishing the connectivity of the final exterior surface triangulation involve computation of the sign of this determinant by design. As a result of this choice, the robustness of the overall procedure ultimately equates to a robust implementation of the signed volume calculation. Fortunately, evaluation of this determinant has long been the subject of study in computational geometry and computer science[28,36,50,80].

When the signed volume in eq. 2.3 (page 46) is computed for arbitrarily positioned geometry, it can return a result which is positive (+1), negative (–1) or zero (0), where ±1 are non-degenerate cases and zero represents some geometric degeneracy. Note that to this point in our discussion, the restriction to generally positioned data protected us from the zero case. Implementation of this predicate, however, can be somewhat delicate, since it requires that we distinguish round-off error from an exact zero. Such considerations usually lead practitioners to implement the predicate with exact arithmetic, either through extended floating-point precision or by integer representations of the data. Unfortunately, while much hardware development has gone into rapid floating point computation, few hardware architectures are optimized for either the arbitrary precision floating-point or integer math alternatives.

2.3.1 Integer Arithmetic

The integer math alternative is quite common in computational geometry and graph theory, and is making inroads in CFD. The approach can be considered an *inflation* strategy, since it begins with the data being "inflated" to span the allowable range of integers. Often, data is then perturbed slightly to its nearest allowable location on the integer grid. The examples in references [36] and [69] rely on integer computations, as do the discussions and examples in reference [64]. Since the integers are a special subset of the reals which can be represented exactly on computational hardware, integer arithmetic is exact for the subset of integers which the hardware (or software) can support. While a simple integer approach may just use the hardware's integer words, real-world examples generally require extended integer representations in software.

As an example of the need for extended integer representations, consider application of the incircle predicate for a 2-D Delaunay triangulation using integer math. On a 32-bit machine, unsigned integers can represent numbers from zero to 4294967296. However, the incircle predicate involves evaluating terms of the form $x_a \cdot x_b \cdot x_c^2$ due the evaluation of the 3 x 3 determinant in eq. 2.3 and the mapping in eq. 2.9 which involves squaring the coordinate data. The product of two p bit integers is exactly representable in $2p - 1$ bits, so the coordinate data in the domain must be representable by only 8 bits. This result restricts the range of allowable input data to the interval [0, 511]. Such low resolution is generally insufficient for real world geometries. A variety of extended resolution integer packages are available, including that

described in reference [23] and others available freely from the internet[1] and commercial sources. The mechanics of expansion schemes for extended precision integer arithmetic are described in references [48,38].

Although they are attractive for implementing purely combinatorial algorithms, integer methods can be restrictive for geometric algorithms which generate geometry. When a constructor generates new data, it must be moved to a topologically consistent position on the integer grid, or represented internally with an extended precision integer representation. Exact computation with extended precision data leads to operations whose expense is dependent upon their context[96]. Thus software arithmetic on long integer expansions can be computationally expensive.

Reference [69] describes an extended integer scheme for unstructured (tetrahedral) mesh generation using expansions of character data types. This approach attempts to avoid extensive use of software arithmetic by restricting input data to 30-bit integer resolution[2]. The method makes extensive use of an *insphere* predicate which computes the signed volume of a simplex in 4 dimensions (see eq. 2.2). Computations are carried out on the double-precision floating-point hardware, unless the result is ambiguous due to extended bit-requirements. Cases which do require extended precision (reportedly 0.01%) are computed using the exact software arithmetic routines, based upon multiple digit expansions using 8-bit digits. The technique of using the floating-point hardware and “trapping” indeterminate cases is an example of *floating-point filtering*. However, in this implementation there is still the drawback that the “given” data must be initially perturbed to the integer grid.

2.3.2 Exact Floating-Point Arithmetic

An alternative to the exact integer arithmetic approach is offered by exact floating-point computation[72,81]. This is also a “software arithmetic” approach, but it has the attraction of not requiring initial data to be perturbed before the computation can begin. The “given” geometry may be specified in floating-point, and the topological predicates are evaluated using the standard floating-point hardware. In this case, one constructs a floating-point filter by comparing the result of eq. 2.3 with an *a-posteriori* estimate of the maximum possible value of the round-off error in the determinant. If this error is larger than the computed signed volume then the case is

1. bigum, longnum, LIA, GNU calc and others.
2. Data is normalized to the interval $[-10^9, 10^9]$ in each coordinate.

considered indeterminate and it is re-evaluated using adaptive precision floating-point arithmetic[1].

3 x 3 or 4 x 4?

The simplex determinant for *d*-dimensional coordinate data (eq. 2.2) consists of the coordinate data in the first *d* columns of the matrix, and a column of 1's in the final column. Linear Algebra reminds us that a determinant of this form may be reduced in order by one. Eq. 2.3 makes use of this and expresses the signed volume of a tetrahedron as either a 3×3 or a 4×4 determinant. Computationally, however, these forms are not equivalent, and the decision is not straightforward[81].

At first glance, it seems apparent that the 3×3 form is preferable because expansion of the determinant involves fewer terms making it less expensive. The subtraction in the 3×3 is menacing, however, since it can destroy the precision of the result even before one begins evaluating the determinant. Therefore it is important to identify the cases where this subtraction will result in round-off error and where it will be exact.

This discussion on binary arithmetic uses the notation $\oplus$, $\ominus$ and $\otimes$ to represent *p*-bit floating-point addition, subtraction and multiplication. Results which cannot be represented exactly with a *p*-bit significand use the *round-to-even* rule[2], which is consistent with the IEEE 754 standard for floating-point operations.

In reference [83], Sterbenz proves that:

$$\text{If} \quad b \in \left[\frac{a}{2}, 2a\right] \text{ then} \quad a \ominus b = a - b \ . \tag{2.10}$$

The floating-point subtraction of two *p*-bit numbers $a \ominus b$ is exact if *b* is between *a*/2 and 2*a*. Intuitively this makes sense if one considers the binary representation of *a* and *b*. If $b \in [a/2, 2a]$, then *a* and *b* have either the same exponent, or their exponents may differ by one. When *a* and *b* have the same exponent, the subtraction is simply subtraction of the mantissa of *a* and *b*. This result is clearly exact, and expressible in *p*-bits. If the exponents differ by one, then the difference has the

1. Our implementation uses the adaptive precision floating-point package of ref. [81], see also [80].

2. "Round-to-even" is a tie-breaking rule, which is specified by the IEEE 754 standard as default. If a multiplication of two numbers yields a value which is halfway between 2 *p*-bit numbers, then the result is rounded to the nearest *p*-bit representation with an even significand. For example, the binary number 11011 would be rounded to 1.110×2^4 using $p = 4$ bits and the round-to-even rule for tie-breaking.

```
    a  =  1  1  0  1          a  =  1  0  0  1    × 2^1
    b  =  1  0  1  0          b  =     1  0  0  1
-------------------------  -------------------------
a - b  =        1  1       a - b  =     1  0  0  1
```

Figure 2-23: Demonstration of exact floating-point subtraction from eq. 2.10. (following Ref.[80]).

smaller of the two exponents, and can still be expressed in p-bits. Figure 2-23 illustrates this result with examples from reference [80].

As a result of the relationship in eq. 2.10, the floating-point subtraction in the 3×3 form of the signed volume computation (eq. 2.3) is exact if the coordinates of the geometry meet the criteria in eq. 2.10. In practice, most surface triangulations consist of objects which are relatively small by comparison with their distance to the origin, so the restriction that $b \in [a/2, 2a]$ is not severe. For illustration, Figure 2-24 shows an example using the surface triangulation on a teardrop shaped body. In the figure, triangles for which floating-point subtraction of their vertices is not exact are shaded. As shown, the only triangles which violate this criterion lie near a coordinate axes, so the vast majority of these triangles may use the 3×3 simplex determinant without incurring round-off error due to the initial coordinate subtraction.

2.3.3 Floating-Point Filtering and Error Bounds

As discussed in the first paragraph of the previous section, the strategy for accurate and robust computation of the orientation test in eq. 2.3 may be outlined as follows:

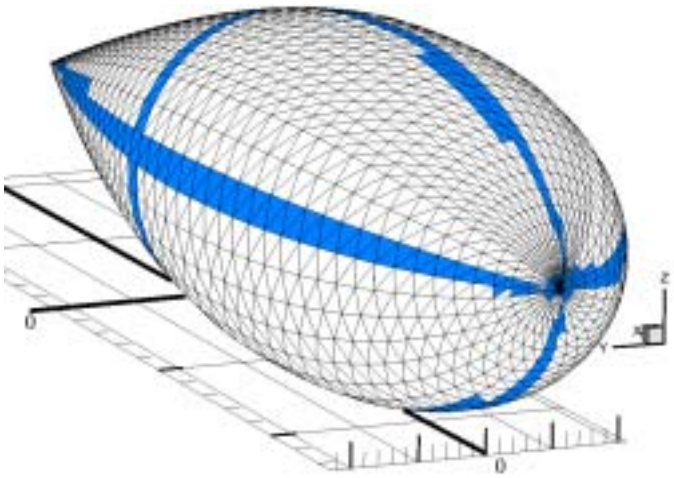


Figure 2-24: The unshaded triangles may use the 3×3 form of the simplex determinant in eq. 2.3 without incurring round-off error due to the initial subtraction of coordinate data.

1. Compute the orientation test (eq. 2.3) using data in floating-point
2. Compute maximum round-off error bound, $\varepsilon_{RE\,max}$, for the floating point evaluation with the data in **1**.
3. If the absolute magnitude of the signed volume from the orientation test is less than the error. bound, recompute the orientation test using exact, adaptive precision floating-point arithmetic.

The maximum error bound in step 2 is the quantity which *filters* out the questionable floating-point computations. Derivation of this error bound is not immediately intuitive, therefore, in this section we sketch the procedure.

Keeping track of the floating-point errors in a computation involves expanding each operation into a result and an error term. For example when the real sum $a + b$ is computed by the floating-point expression $a \oplus b$, the exact result expands into (x, y),where x is the approximate value of the sum and y is the round-off error which remains bounded. The magnitude of the round-off error for $x \Leftarrow a \oplus b$ can be no greater than $\varepsilon|x|$, and must also be smaller than $\varepsilon|a + b|$. *Machine epsilon*, ε, is precisely defined since it comes from the rounding in the last bit of the significand. Using p-bit significands,

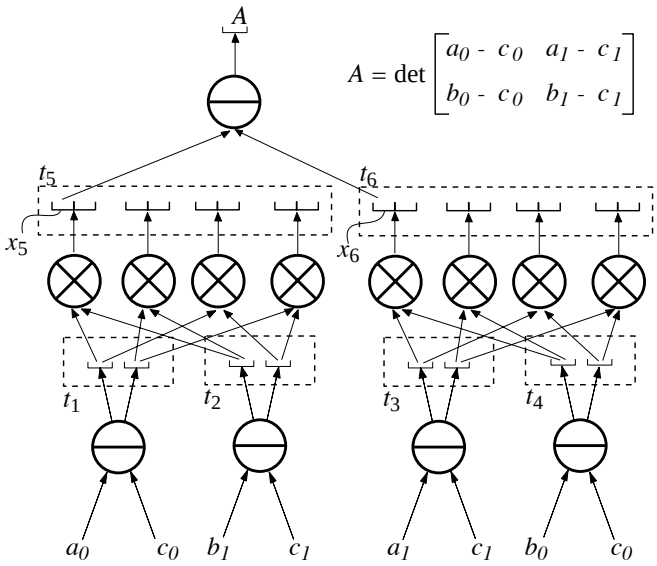
$$\varepsilon = 2^{-p}. \tag{2.11}$$

On IEEE 754 compliant platforms, ε is 2^{-24} in single precision, and 2^{-53} in double. One may check this on any platform by determining the largest exponent for which $1.0 \oplus 2^{-p} = 1.0$ when both the sum and the equality are evaluated in floating-point.

To illustrate the procedure for deriving these error bounds, consider the 2-D form of the orientation test. In two dimensions, the orientation test of eq. 2.3 may be expressed with the following determinant.

$$A = \det\begin{pmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{pmatrix} = \det\begin{pmatrix} a_0 - c_0 & a_1 - c_1 \\ b_0 - c_0 & b_1 - c_1 \end{pmatrix} \tag{2.12}$$

Figure 2-25 shows an expression tree for the evaluation of this determinant. Following reference [80], the expansions of sub-expressions in this tree are labeled with a t for “true”. If t represents the true result of any expression $a + b = t$, and x is the result from floating-point, $x = a \oplus b$, then the error satisfies both $t = x \pm \varepsilon|x|$ and $t = x \pm \varepsilon|t|$. With this notation, if t_i is the true result for any sub-expression i in the figure, x_i is its approximate counterpart.



True Values	Approximate Value
$t_1 = a_0 - c_0$	$x_1 = a_0 \ominus c_0$
$t_2 = b_1 - c_1$	$x_2 = b_1 \ominus c_1$
$t_3 = a_1 - c_1$	$x_3 = a_1 \ominus c_1$
$t_4 = b_0 - c_0$	$x_4 = b_0 \ominus c_0$
$t_5 = t_1 t_2$	$x_5 = x_1 \otimes x_2$
$t_6 = t_3 t_4$	$x_6 = x_3 \otimes x_4$
$t_A = t_5 - t_6$	$x_A = x_5 \ominus x_6$

Figure 2-25: Expression tree for 2-D orientation predicate in eq. 2.21

Figure 2-25 shows a list of approximate and true values in the expression tree. Floating-point results have error associated with them, and this error is bounded by ε scaled by the magnitude of the floating point result. For example the floating-point product $x_5 = x_1 \otimes x_2 = x_1 x_2 \pm \varepsilon |x_5|$. From this, we can express each true term as an approximate term and an associated error term.

$$\begin{aligned} t_5 = t_1 t_2 &= (x_1 \pm \varepsilon|x_1|)(x_2 \pm \varepsilon|x_2|) \\ &= x_1 x_2 \pm (2\varepsilon + \varepsilon^2)|x_1 x_2| \\ &= x_5 \pm \varepsilon|x_5| \pm (2\varepsilon + \varepsilon^2)(|x_5| \pm \varepsilon|x_5|) \\ &= x_5 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_5| \end{aligned} \tag{2.13}$$

The true value t_6 plays the same role on the right side of the expression tree, giving:

$$t_6 = x_6 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_6| \tag{2.14}$$

With t_5 and t_6 complete, there is only one floating-point subtraction left at the top of the expression tree.

$$\begin{aligned} t_A = t_5 - t_6 &= x_5 - x_6 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \\ &= A \pm \varepsilon|A| \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \end{aligned} \tag{2.15}$$

With eq. 2.5, the error bound is nearly in hand, only the $\pm\varepsilon|A|$ remains ambiguous. However, recall that we are only interested in the sign of A. Taking advantage of this fact, the sign of A must be correct, provided that

$$\begin{aligned} A &> \pm\varepsilon|A| \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \\ (1-\varepsilon)|A| &> (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \end{aligned} \tag{2.16}$$

Dividing through by $(1-\varepsilon)$ gives the bound on $|A|$.

$$|A| \geq (3\varepsilon + 6\varepsilon^2 + 8\varepsilon^3)(|x_5| + |x_6|) \tag{2.17}$$

Going from eq. 2.16 to eq. 2.17 involves expanding in series for the division by $(1-\varepsilon)$, this series has been truncated at ε^3 and rounded up in eq. 2.17, which converts the "$>$" to the "$\geq$" shown. As it stands in eq. 2.17, the error bound on A is not yet directly applicable. Computation of the bound itself will incur an error of $(1-\varepsilon)$ for the addition of $|x_5|$ and $|x_6|$, and another factor of $(1-\varepsilon)$ for the product of the two terms on the right side. Converting these two exact operations to their approximate counterparts therefore multiplies the coefficient in eq. 2.17 by $(1-\varepsilon)^2$. Thus, the sign of A is guaranteed to be correct if

$$|A| \geq (3\varepsilon + 12\varepsilon^2 + 24\varepsilon^3) \otimes (|x_5| \oplus |x_6|)\,. \tag{2.18}$$

Unfortunately, the coefficient in this expression is not exactly expressible in p bits, so we must round up to the next p bit number, This gives the final expression for the error bound on the 2-D simplex determinant.

$$|A| \geq (3\varepsilon + 16\varepsilon^2) \otimes (|x_5| \oplus |x_6|) \tag{2.19}$$

The coefficient $(3\varepsilon + 16\varepsilon^2)$ must be computed (exactly) once at the beginning of the computation, and may then be applied to the computed value of A by multiplying with the sum of the minors $|x_5|$ and $|x_6|$ all using floating-point.

The right side of eq. 2.19 is the maximum value of the round-off error in computing the 2-D simplex determinant in eq. 2.12. This bound accounts for not only possible error in computing the determinant, but also for the error associated with computing the bound itself.

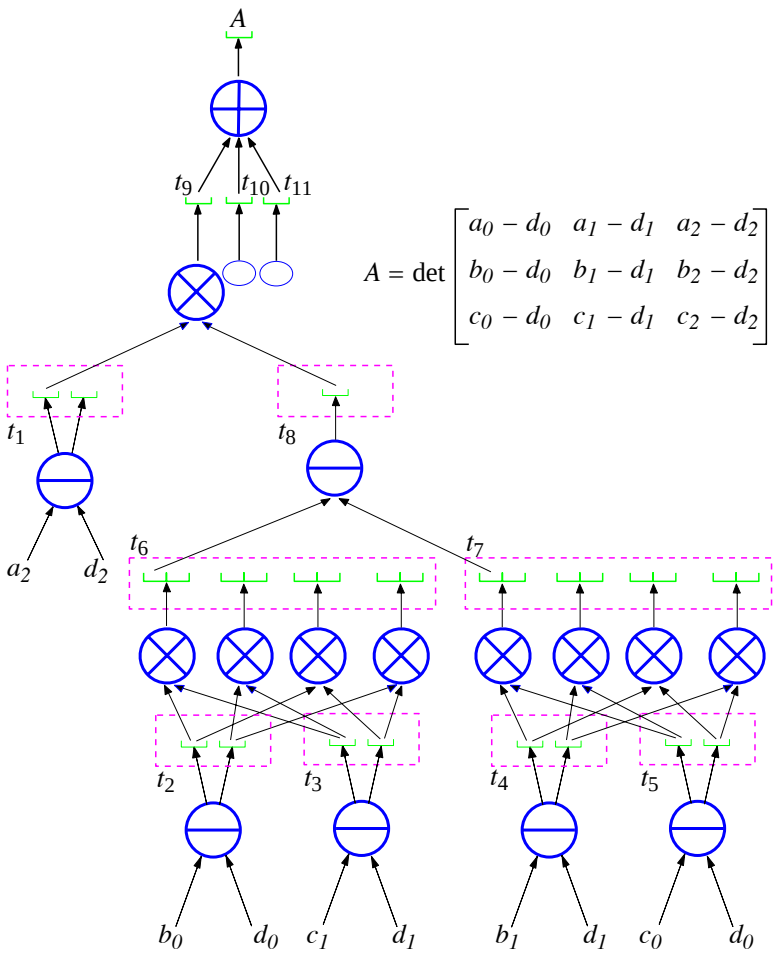


Figure 2-26: Expression tree for three dimensional orientation test of eq. 2.3, which computes the signed volume of the tetrahedron $T_{a,b,c,d}$.

Just as this bound was computed by examining the expression tree for the 2-D orientation test, examination of the expression tree for the 3-D orientation test of eq. 2.3 permits us to derive an error bound for this predicate as well. Figure 2-26 shows the expression tree for the 3-D predicate, which evaluates eq. 2.3 in its 3×3 form.

The error bound, $\varepsilon_{RE\,max}$, for floating-point computation of the 3-D orientation test may be computed following a derivation similar to that presented in the preceding paragraphs, but with slightly more work. This error bound is:

$$\varepsilon_{REmax} = (7\varepsilon + 56\varepsilon^2) \otimes (\alpha_A \oplus \alpha_B \oplus \alpha_C) \tag{2.20}$$

Where the second term on the right side, $(\alpha_A \oplus \alpha_B \oplus \alpha_C)$, is the permanent of the 3×3 matrix in eq. 2.3 (and Figure 2-26) evaluated in floating-point. These terms are:

$$\alpha_A = \left|a_2 \ominus d_2\right| \otimes \left(\left|(b_0 \ominus d_0) \otimes (c_1 \ominus d_1)\right| \oplus \left|(b_1 \ominus d_1) \otimes (c_0 \ominus d_0)\right|\right)$$

$$\alpha_B = \left|b_2 \ominus b_2\right| \otimes \left(\left|(c_0 \ominus d_0) \otimes (a_1 \ominus d_1)\right| \oplus \left|(c_1 \ominus d_1) \otimes (a_0 \ominus d_0)\right|\right) \quad (2.21)$$

$$\alpha_C = \left|c_2 \ominus d_2\right| \otimes \left(\left|(a_0 \ominus d_0) \otimes (b_1 \ominus d_1)\right| \oplus \left|(a_1 \ominus d_1) \otimes (b_0 \ominus d_0)\right|\right)$$

If the magnitude of the signed volume of the tetrahedron $T_{a,b,c,d}$ is less than $\varepsilon_{RE\,max}$ from eq. 2.20, then the predicate must be evaluated using exact arithmetic. The applications referred to in these notes use the package in reference [81].

In practice, only a very small fraction of the determinant evaluations ever get trapped by this filter. For example, in the helicopter configuration shown earlier in Figure 2-21, the intersection required 1.37M evaluations of the determinant, and of these, only 68 (0.005%) failed to pass the floating point filter of eqs.2.20 and 2.21.

2.4 Tie-Breaking, Degeneracy and Virtual Perturbations

In expanding our discussion to permit degenerate data, its important to consider how the orientation test in eq. 2.3 and the exact arithmetic will handle such cases. The orientation test computes the signed volume of a tetrahedron $T_{a,b,c,d}$. Degenerate data result in a signed volume of identically zero, which implies that all four test points are coplanar. This situation is referred to as a *tie*. With the exact arithmetic routines in place, ties can be reliably identified, and so focus shifts to the topic of *tie-breaking*. This section lifts the assumption of generally positioned data and we now consider arbitrarily positioned geometry.

The richness of possible geometric degeneracies in three dimensions cannot be overstated, and without some systematic method of identifying and coping with them, handling of special cases can permeate, or even dominate the design of a geometric algorithm[26]. Rather than attempt to implement an *ad-hoc* tie-breaking algorithm based on intuition and programmer skill, we seek an algorithmic approach to this problem.

Simulation of Simplicity (SoS) is one of a category of general approaches to degenerate geometry known generically as *virtual*, or *symbolic perturbation* algorithms[36,94,95]. The basic premise is to imagine that all input data undergoes a unique, ordered perturbation such that all ties are broken (*i.e.* data in special posi-

tion is perturbed into general position). When a tie is encountered, we rely on this set of virtual perturbations to break the tie. Since the perturbations are both unique and constant, any tie in the input geometry will always be resolved in a topologically consistent manner. Since the perturbations are virtual, no given geometric data is ever altered. When there is no tie, then the perturbations have no effect on the outcome.

The perturbation $\varepsilon(i, j)$ at any point is a function of the point's index, $i \in \{0, 1, \ldots, N-1\}$ and the coordinate direction, $j \in \{1, \ldots, d\}$. Various researchers have suggested perturbations of different forms, but here we consider that in reference [36]:

$$\varepsilon(i, j) = \varepsilon^{2^{i \cdot \delta - j}} \qquad \text{where} \quad \begin{matrix} 0 \le i \le N-1 \\ 1 \le j \le d \\ \delta \ge d \end{matrix} \tag{2.22}$$

This choice indicates that the perturbation applied to i_j is always greater than that on k_l iff $(i < k)$ or $(i = k) \wedge (j > l)$. This means that points with lower indices are always perturbed more than points with higher indices, and on any one point, the perturbation on the data of the lower coordinate directions is greater than that on higher coordinate directions.

Figure 2-27 shows an example of this perturbation applied to a ray-casting problem in 2-D to help demonstrate. In this example, a ray cast from pt. 5 in the original configuration intersects improperly with segments $\overline{36}$, $\overline{47}$, $\overline{78}$, and $\overline{12}$, and is colinear with $\overline{64}$ and $\overline{81}$. If a perturbation of the form in eq. 2.22 were applied, these degener-

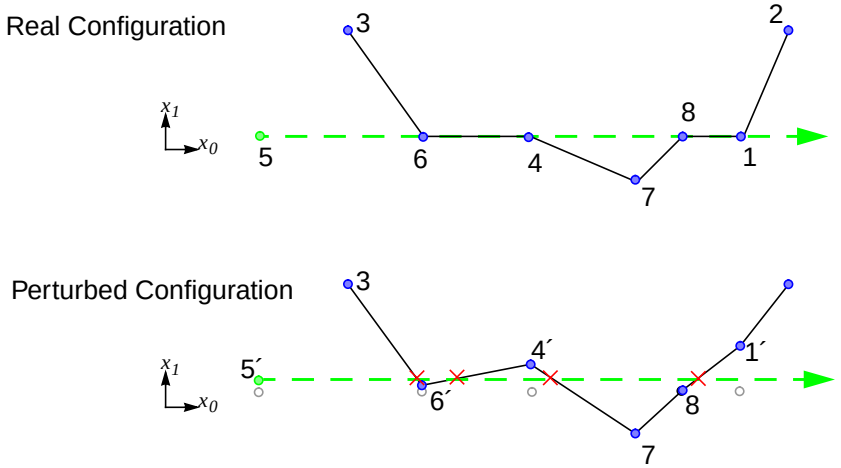


Figure 2-27: Degeneracy breaking by virtual perturbation using the perturbation of eq. 2.22. For clarity, perturbations only applied to the vertical (x_1) coordinate.

acies would be resolved as shown in the lower part of the figure. Note that for clarity, the data in the figure have only been perturbed in the vertical direction.

To illustrate the application of this scheme, consider again the two dimensional version of the simplex determinant in eq. 2.2.

$$\det[M] = \det\begin{pmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{pmatrix} \tag{2.23}$$

If the points *a,b,c* are assumed to be indexed with $i = 0, 1, 2$ respectively, then taking $\delta = 2$ gives a perturbation matrix with:

$$\Lambda = \begin{pmatrix} \varepsilon^{2^{-1}} & \varepsilon^{2^{-2}} & 1 \\ \varepsilon^{2^{2-1}} & \varepsilon^{2^{2-2}} & 1 \\ \varepsilon^{2^{4-1}} & \varepsilon^{2^{4-2}} & 1 \end{pmatrix} \tag{2.24}$$

Taking the determinant of the perturbed matrix $M_\Lambda = M + \Lambda$ yields:

$$\begin{aligned} \det[M_\Lambda] = \det[M] &+ \varepsilon^{1/4}(-b_0 + c_0) \\ &+ \varepsilon^{1/2}(b_1 - c_1) + \varepsilon^{1}(a_0 - c_0) \\ &+ \varepsilon^{3/2}(1) + \varepsilon^{2}(-a_1 + c_1) \\ &+ \varepsilon^{9/4}(-1) + \ldots \end{aligned} \tag{2.25}$$

Since the data, *a,b,c* span a finite region in 2-space, intuitively one can always envision a perturbation small enough such that increasing powers of ε always lead to terms with decreasing magnitude. Ref.[36] proves that this observation holds for a perturbation of the form of eq. 2.22. If $\det[M]$ ever evaluates to an exact zero, the sign of the determinant will be determined by the sign of the next significant coefficient in the ε expansion. If the next term also yields an exact zero, we continue checking the signs of the coefficients until a non-zero term appears. In eq. 2.25 the coefficient on the fifth term ($\varepsilon^{3/2}$) is a constant (–1) and since *sign*(-1) is always negative, this term will never be degenerate.

Returning momentarily to the sketch in Figure 2-27, take points 4, 5, and 6 as points *a, b,* and *c,* (respectively) in the determinant matrix of eq. 2.23. In this case, $\det[M]$ would be exactly zero since all three points are colinear, and triangle $\Delta_{4,5,6}$ has zero

area. However, the first term ($\varepsilon^{1/4}$) in the ε expansion of eq. 2.25 dictates that this degeneracy may be resolved by checking ($-b_0 + c_0$). Since point 6 is to the right of point 5, this result is positive (+). Thus, $\Delta_{4',5',6'}$ is taken as having a positive area. This implies that the tie-breaking scheme perceives point 6′ as lying below a line segment joining 4′ to 5′, which is consistent with the sketch of Figure 2-27 for the perturbed configuration.

The three dimensional variant of the simplex determinant (eq. 2.3) has 15 non-zero coefficients before the first constant is encountered. Table 2.1 lists the hierarchy of terms in the 3-D expansion, which is analogous to that given in eq. 2.25.

Table 2.1. The first 15 non-zero coefficients in the expansion of the 3-D simplex determinant, listed in increasing powers of ε.

Term	exponent	coefficient
1	ε^0	$\det\begin{pmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{pmatrix}$
2	$\varepsilon^{1/8}$	$\det\begin{pmatrix} b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
3	$\varepsilon^{1/4}$	$(-1)\det\begin{pmatrix} b_0 & b_2 & 1 \\ c_0 & c_2 & 1 \\ d_0 & d_2 & 1 \end{pmatrix}$
4	$\varepsilon^{1/2}$	$\det\begin{pmatrix} b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \\ d_1 & d_2 & 1 \end{pmatrix}$
5	ε^{1}	$(-1)\det\begin{pmatrix} a_0 & a_1 & 1 \\ c_0 & c_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
6	$\varepsilon^{5/4}$	$\det\begin{pmatrix} c_0 & 1 \\ d_0 & 1 \end{pmatrix}$
7	$\varepsilon^{3/2}$	$(-1)\det\begin{pmatrix} c_1 & 1 \\ d_1 & 1 \end{pmatrix}$

Table 2.1 (Continued)

Term	exponent	coefficient
8	ε^{2}	$\det\begin{pmatrix} a_0 & a_2 & 1 \\ c_0 & c_2 & 1 \\ d_0 & d_2 & 1 \end{pmatrix}$
9	$\varepsilon^{5/2}$	$\det\begin{pmatrix} c_2 & 1 \\ d_2 & 1 \end{pmatrix}$
10	ε^{4}	$(-1)\det\begin{pmatrix} a_1 & a_2 & 1 \\ c_1 & c_2 & 1 \\ d_1 & d_2 & 1 \end{pmatrix}$
11	ε^{8}	$\det\begin{pmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
12	$\varepsilon^{33/4}$	$(-1)\det\begin{pmatrix} b_0 & 1 \\ d_0 & 1 \end{pmatrix}$
13	$\varepsilon^{17/2}$	$\det\begin{pmatrix} b_1 & 1 \\ d_1 & 1 \end{pmatrix}$
14	ε^{10}	$\det\begin{pmatrix} a_0 & 1 \\ d_0 & 1 \end{pmatrix}$
15	$\varepsilon^{21/2}$	(+1)

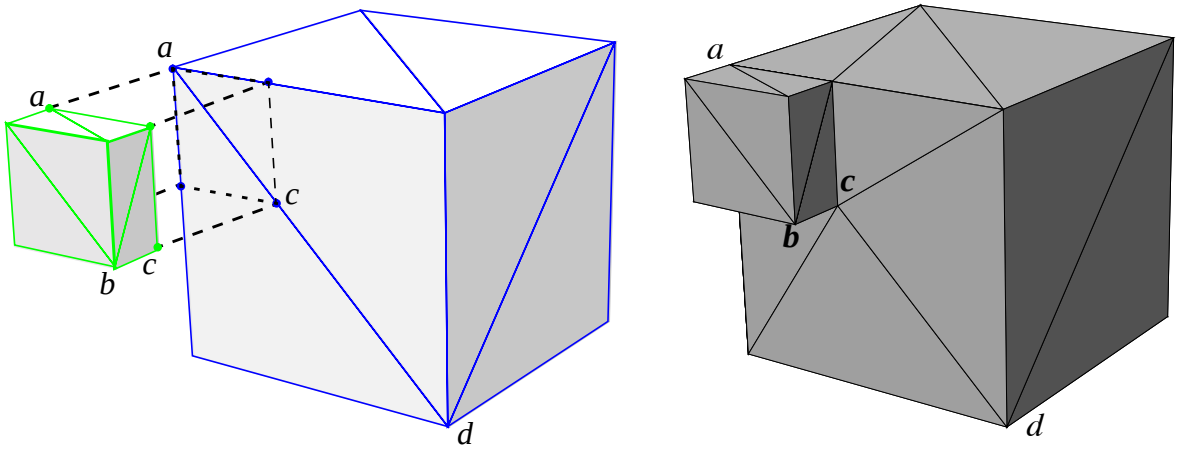


Figure 2-28: Two improperly intersecting right parallelepipeds with degeneracies resolved using virtual perturbations and exact arithmetic. (left) Sketch of components before intersection showing degeneracy. (right) Result after intersection and tie-breaking.

Figure 2-28 contains a deceptively simple looking application of the tie-breaking algorithm. The large and small cubes in the sketch abut against each other exactly. In addition to sharing the same geometry at location *a*, the cubes not only have three coplanar faces, but also have exact improper intersections where edge *bc* abuts against *ad* and elsewhere. The figure shows the result after computing the intersection, re-triangulating, and extracting the wetted surface. The virtual perturbation scheme resolved these degeneracies by imposing virtual perturbations such that the two polyhedra overlapped properly, consistently resolving not only the coplanar degeneracy, but also all improper edge-edge intersections. This geometry required 504 evaluations of eq. 2.3, 186 of which evaluated to exactly zero and required tie-breaking by the virtual perturbation scheme.

The exact arithmetic and tie-breaking routines may be implemented as “wrappers” for the topological primitive in eq. 2.3. Thus the intersection, triangulation and ray-casting algorithms remain unmodified, but now have been extended to consistently treat geometry in special position. With this extension, the entire intersection algorithm can now be robustly applied to any arbitrary geometry, even those with degeneracies.

3. Volume Meshing and Cut-Cells

Generation of the Cartesian volume mesh begins with the wetted surface of the configuration extracted using the intersection process just discussed. Since there is no longer any internal geometry in the surface description, the mesh generation task has been substantially simplified. In general, Cartesian volume meshing is a very straightforward process of nested cell division. Thus, the real technical challenge in the design of an algorithm is to insure that it is as efficient, in both complexity and memory usage, as possible. The volume meshing process will necessarily introduce cut-cells into the domain, and the description of these cut-cells is closely linked to the level of fidelity that one wishes to employ in applying boundary conditions for the flow solver. Thus, in addition to focusing on generation of the final volume mesh, we also present several levels of boundary condition modeling and outline the geometric operations required to support them.

The mesh generation strategy outlined in this section views the mesh as an unstructured collection of Cartesian hexahedra (as in §1.4.3, page 20). While the alternative approaches using octree or structured patches are also attractive, the unstructured approach more readily preserves the ability to directionally refine the mesh cells. This flexibility can be important since research has suggested that permitting only isotropic refinement in three dimensions may lead to excessive numbers of cells for geometries with many length scales and high aspect ratio components[3].

3.1 Counting Arguments and Anisotropic Cell Division

To demonstrate the importance of the ability to anisotropically divide Cartesian cells in three dimensional meshes, consider the example shown in Figure 3-1. Shown is a 9 level adapted mesh and isobars in the discrete solution for a NACA 0012 at $M_\infty = 0.8$ and $\alpha = 1.25^\circ$. This solution is characterized by a strong shock on the lee surface, and a weak shock on the windward side. This example has been widely com-

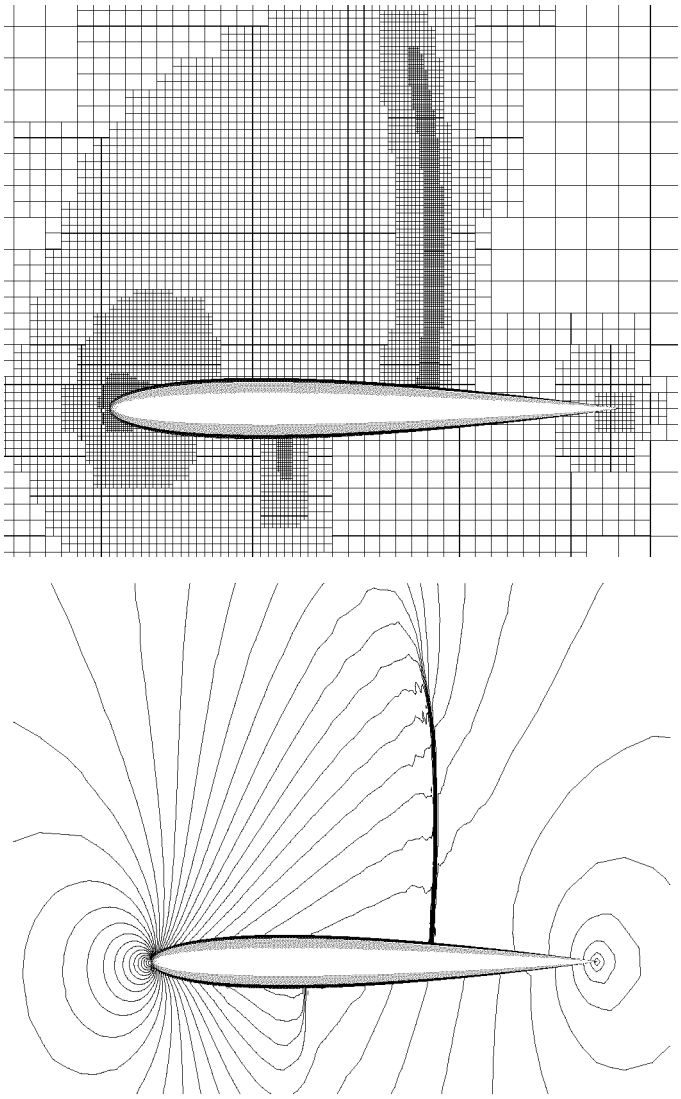


Figure 3-1: Adapted mesh and isobars of discrete solution for unit span NACA 0012 at Mach 0.8 and 1.25˚ angle of attack. The 9 level adapted mesh has 1108839 total cells. The 2D slice shown passes through 14280 cells.

puted in the literature[4]. With 9 levels of isotropic adaptation, the mesh scale at the leading edge is $\Delta x = 0.39\%C$. Figure 3-2 compares the pressure coefficient distribution for this numerical solution with a reference solution from the literature[4]. With 9 levels of mesh refinement, the solutions compare well, indicating that the resolution is adequate for this case.

Although the mesh shown in Figure 3-1 appears 2-D, this example was actually computed as a two dimensional flow over a three dimensional a unit span wing based on the NACA 0012 profile. The mesh and solution shown are the projection of the full 3-D solution on a 2-D cut through the mesh. The 2-D slice passes through 14240 cells,

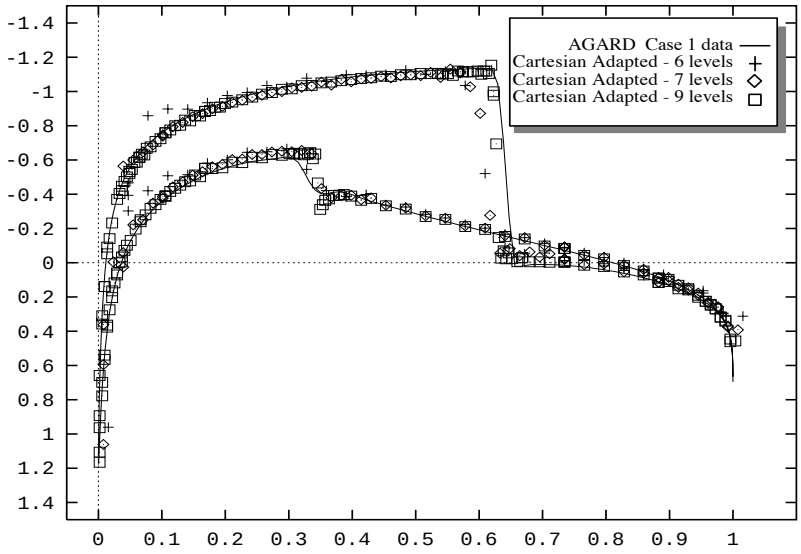


Figure 3-2: *Cp* vs. *x*/*C* for the unit span NACA 0012 example at Mach 0.8 and 1.25˚ angle of attack. Cartesian data plotted using a piecewise constant extrapolation from the centroid of the wall-cut Cartesian cell.

but 9 levels of isotropic refinement resulted in a 3-D mesh with over 1.1M total cells. While 14240 cells is reasonable for the two dimensional case, 1.1M cells is certainly excessive for a unit span wing computation. Since the adaptation was isotropic, each mesh refinement was forced to increase spanwise resolution as well. In the final mesh, there are 256 3-D Cartesian cells for every fine mesh cell shown in the 2-D mesh slice of Figure 3-1. Such excessive resolution can make Cartesian methods uncompetitive as compared to body-fitted approaches.

The result in this example is certainly not unexpected. In typical structured or unstructured computations, one generally uses high aspect ratio cells to increase the resolution of some directions while leaving others relatively coarse. For Cartesian mesh methods, this equates to permitting anisotropic cell division during mesh generation and adaptation.

In reference [3], examples such as this were used to frame *counting arguments* for Cartesian mesh methods. These arguments suggested that in order to avoid excessive numbers of final cells in three dimensions, Cartesian methods must necessarily employ anisotropic cell division. The unit span example in Figures 3-1 and 3-2 is actually relatively mild. More extreme demonstrations of these counting arguments come from considering geometry with high aspect ratio components like a wing flap, or a flap-vane. Such components frequently have aspect ratios in the range of 10-20,

and thus, proper streamwise resolution of these components makes anisotropic cell division a necessity.

3.2 Volume Mesh Generation

We wish to capitalize on the simplicity of nested Cartesian meshes to produce an algorithm which has linear asymptotic complexity. This aim is a reasonable since the mesh geometry is well defined, and mesh locations may be generated by bisection of parent cells, using only local information. The other main goal of this section is to ensure that the scheme uses memory efficiently. The coordinate-aligned nature of Cartesian meshes suggest many simplifications which the algorithm takes advantage of in order to produce an efficient scheme.

3.2.1 Proximity Testing

Let $\{T\}$ be the set of triangles describing the wetted surface of the configuration that was constructed by the intersection algorithm in section 2. If, the N_T surface triangles in $\{T\}$ are inserted into an ADT, then locating the subset T_i of triangles actually intersected by the i^{th} Cartesian cell will have complexity proportional to $\log(N_T)$. When a cell is subdivided, a child cell inherits the triangle list of its parent. As the mesh subdivision continues, the triangle lists connected to a surface intersecting ("cut") Cartesian cell will get shorter by approximately a factor of 4 with each successive subdivision. Figure 3-3 illustrates the passing of a parent cell's triangle list to its children.

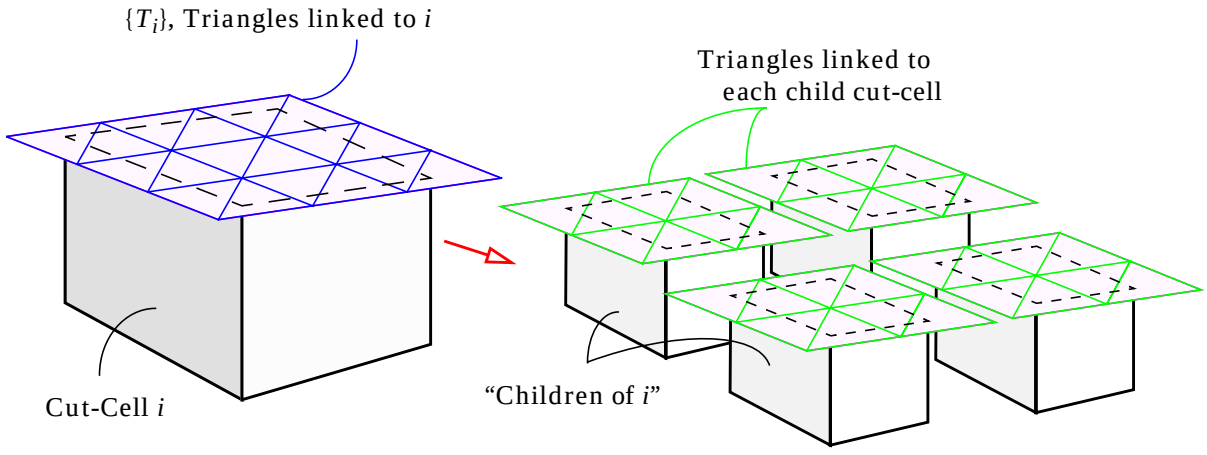


Figure 3-3: List of triangles associated with children of a cut-cell may be obtained using ADT, or by exhaustively searching over the parent cell's triangle list.

This observation that implies that there is a machine dependent crossover beyond which it becomes faster to simply perform an exhaustive search over a parent cell's triangle list, rather than perform an ADT lookup to get a list of intersection candidates for cell *i*. This is easy to envision since all of the triangles that are linked to a child cut-cell must have originally been members of the parent cell's triangle list. If a parent cell intersects only a very small number of triangles, then there is no reason to perform a full intersection check using the ADT. The level of this crossover is primarily determined by the number of elements in N_T and the processor's instruction cache size. For the example problems with $O(10^4)$ to $O(10^6)$ triangles, the crossover from ADT to exhaustive lookup typically occurs for cells with about $2000 < N_{T_i} < 5000$.

As the cells continue to refine, cut-cells are linked to shorter and shorter triangle lists. By conducting searches over the parent's triangle list we take advantage of this fact, and progressively smaller Cartesian cells can be intersected against T with ever decreasing computational complexity.

3.2.2 Geometric Refinement

All surface intersecting Cartesian cells in the domain are initially automatically refined a specified number of times $(R_{min})_j$. Typically, this level is set to be 4 divisions less than the maximum allowable number of divisions $(R_{max})_j$ in each direction. When a cut cell is tagged for division, the refinement is propagated several (usually 3-5) layers into the mesh using a "buffering" algorithm which operates by sweeps over the faces of the cells.

Further refinement is based upon a curvature detection strategy similar to that originally presented in reference [56]. This is a two-pass strategy which first detects angular variation of the surface normal, $\hat{n}$, within each cut cell and then examines the average surface normal behavior between two adjacent cut cells.

Taking k as a running index to sweep over the set of triangles, T_i, V_j is the j^{th} component of the vector subtraction between the maximum and minimum normal vector components in each Cartesian direction.

$$V_j = \max(n_{k_j}) - \min(n_{k_j}) \qquad \forall\, k \in T_i \tag{3.1}$$

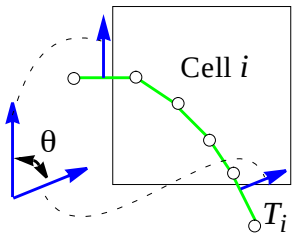


Figure 3-4.a: Measurement of the maximum angular variation within cut-cell *i*.

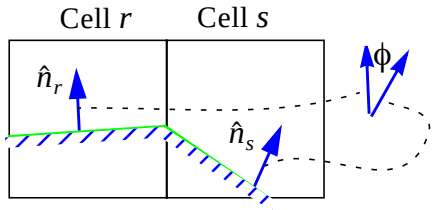


Figure 3-4.b: Measurement of the angular variation between adjacent cut-cells.

The min(-) and max(-) are performed over all elements of T_i. The angular variation within cell *i* is then simply the direction cosines of $\bar{V}$.

$$\cos(\theta_{i_j}) = \frac{V_j}{|\bar{V}|} \tag{3.2}$$

Similarly, $(\phi_j)_{r,s}$ measures the j^{th} component of the angular variation between any two adjacent cut cells *r* and *s*. With $\hat{n}_i$ denoting the unweighted unit normal vector within any cut cell *i*, the components of $\bar{\phi}_{r,s}$ are:

$$\cos(\phi_j)_{r,s} = \frac{|n_{j_r} - n_{j_s}|}{|\hat{n}_r - \hat{n}_s|} \tag{3.3}$$

If θ_j or ϕ_j in any cell exceeds a preset angle threshold then the offending cell is tagged for subdivision in direction *j*. Figures 3-4.a and 3-4.b illustrate the construction of ϕ and θ in two dimensions.

Obviously, by varying ϕ and θ, one may control the number of cut-cells which are tagged for geometric refinement. When ϕ and θ are identically 0˚, all the cut cells will be tagged for refinement, and when they are 180˚ only those at sharp cusps will be tagged. Figure 3-5 explores the sensitivity of the refinement to variation of these parameters for angles ranging from 0˚ to 179˚ on three example configurations. For the angle thresholds on the abscissa, the ordinate lists the number of cells which will be in the resulting mesh after refinement (and adding 4 buffer layers). Three configurations are considered, the first is a HWT geometry with its high-lift system deployed (see Figure 1-16), the second is a wing-fuselage (only) configuration constructed of the fuselage and wing from this same aircraft, and the third is a High Speed Supersonic Transport (HSCT) configuration discussed in section 3.4. In all three cases, varying the angle thresholds for geometric refinement is shown to vary

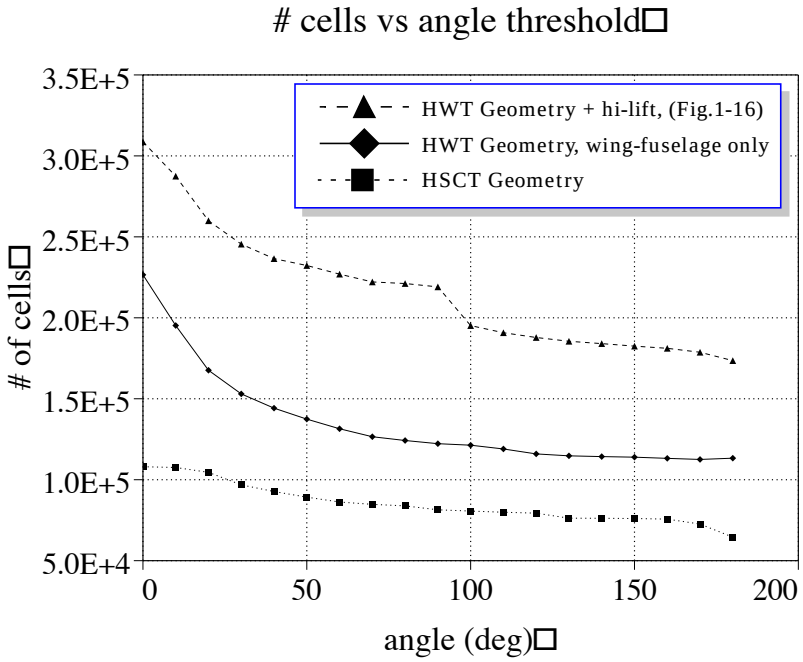


Figure 3-5: Effect if varying angle threshold on number of cells produced after refinement for three different configurations.

the number of cells in the final mesh by approximately 50%, with most of the variation occurring before 50˚. In practice, these thresholds are generally set near 25˚.

3.2.3 Data Structures

The domain for a coordinate aligned Cartesian mesh may be defined by its minimum and maximum coordinates $\bar{x}_o$ and $\bar{x}_1$. Initially, this region is uniformly discretized with N_j divisions in each Cartesian dimension, $j = \{0, 1, \ldots, d-1\}$. The refinement criteria of the previous section provide a method for identifying Cartesian cells near features in the geometry, and by repeatedly dividing these cells and their neighbors a final, geometry adapted mesh is obtained. Since we have adopted an unstructured approach and intend to construct meshes with 10^6 or 10^7 cells, its imperative that the data structures storing the mesh be as compact as possible. The system described in this section provides all cell geometry and cell-to-vertex pointers in 96 bits.

Integer Coordinates

Figure 3-6 shows a model of the j^{th} direction of a Cartesian mesh covering the region $[\bar{x}_0, \bar{x}_1]$. As shown in the sketch, specifying the domain with $\bar{x}_0$ and $\bar{x}_1$ and the initial partitioning by N_j, uniquely identifies all possible Cartesian cell locations in this

region. Each additional refinement increases the maximum integer coordinate by a factor of $2(N_j - 1)$. This relationship suggests a natural mapping to a system of integer coordinates. If one defines a maximum number of permissible cell divisions in this direction, $Rmax_j$, then any point in such a mesh can be uniquely located by its integer coordinates (i_0, i_1, i_2). If we allocate m bits of memory to store each integer i_j, the upper bound on the permissible total number of vertices in each coordinate direction is 2^m.

Figure 3-6 demonstrates that on a mesh with N_j prescribed nodes, performing R_j cell refinements in each direction will produce a mesh with a maximum integer coordinate of $2^{R_j}(N_j - 1) + 1$ which must be resolvable in m bits.

$$2^{R_j}(N_j - 1) + 1 \le 2^m \tag{3.4}$$

Thus, the maximum number of cell subdivisions that may be addressed by a set of m-bit integer coordinates is:

$$(R_{max})_j = \left\lfloor \log_2(2^m - 1) - \log_2(N_j - 1) \right\rfloor \tag{3.5}$$

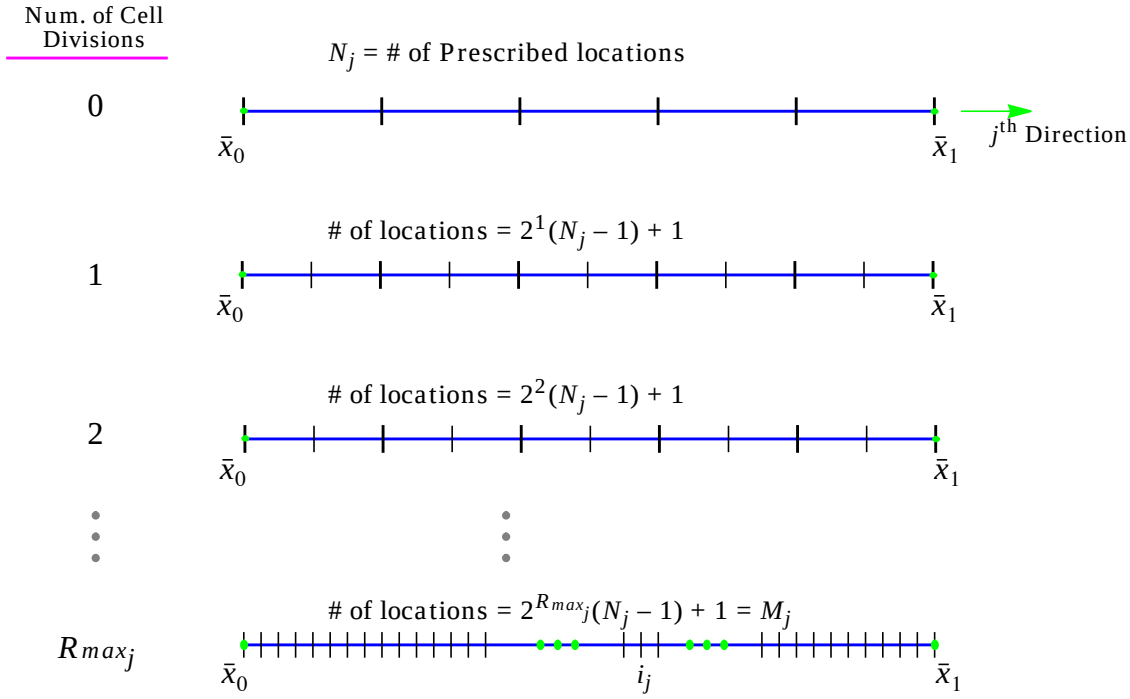


Figure 3-6: Specification of integer coordinate locations for a coordinate direction with N_j prescribed locations.

where the floor "⌊ ⌋" indicates rounding down to the next lower integer. Substituting back into eq. 3.4 gives the total number of vertices which we can address in each coordinate direction using m-bit integers and with N_j prescribed nodes in the direction.

$$M_j = 2^{R_{max_j}}(N_j - 1) + 1 \tag{3.6}$$

Thus, the floor in eq. 3.5 insures that M_j can never exceed 2^m. Figure 3-7 illustrates the integer coordinate numbering scheme in three dimensions.

With the maximum integer coordinate of each direction defined, the geometric location, $\bar{x}$, of the point with integer coordinates $\bar{i}$ may be expressed with a simple vector relation.

$$\bar{x} = \bar{x}_0 + \frac{\bar{i}}{\bar{M}}(\bar{x}_1 - \bar{x}_0) \tag{3.7}$$

The examples in these notes use up to $m = 21$ bits per direction which gives about 2.1×10^6 addressible locations in each coordinate direction. This choice has the advantage that all three indices may then be packed into a single 64-bit integer for

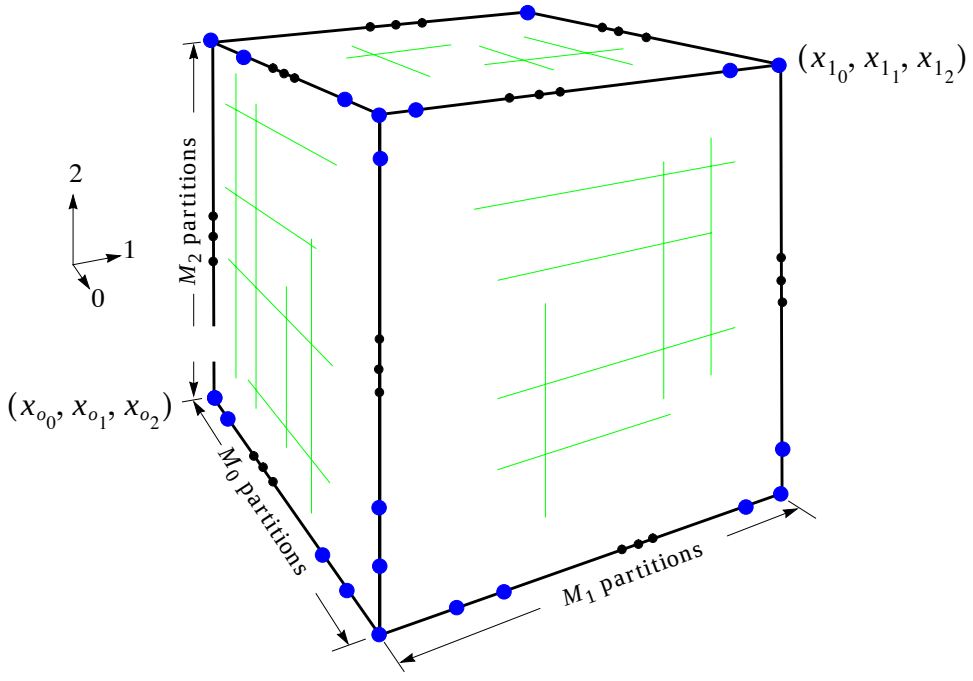


Figure 3-7: Cartesian mesh with M_j total divisions in each direction discretizing the region from $\bar{x}_o$ to $\bar{x}_1$.

storage[1]. Since most 32-bit architectures support 64-bit integers[2], comparisons between integer coordinates may then be performed with a single operator, rather than one operator for each integer coordinate.

Cell-to-Node Pointers

Figure 3-8 gives an example of the vertex numbering within an individual Cartesian cell. This system has been adopted by analogy to the study of crystalline structures specialized for cubic lattices[3]. Within this framework, the cell vertices are numbered with a boolean index of 0 (low) or 1 (high) in each direction. Following this ordering, Figure 3-8 shows the crystal direction of each vertex in square brackets (with no commas). Reinterpreting this 3-bit pattern as an integer yields a unique numbering scheme (from 0-7) for each vertex on the cell.

For any cell i, $\overline{V}_0$ is the integer position vector $(V_{0_0}, V_{0_1}, V_{0_2})$ of its vertex nearest to the $\overline{x}_0$ corner of the domain. If we also know the number of times that cell i has been divided in each direction, R_j, we can express its other 7 vertices directly.

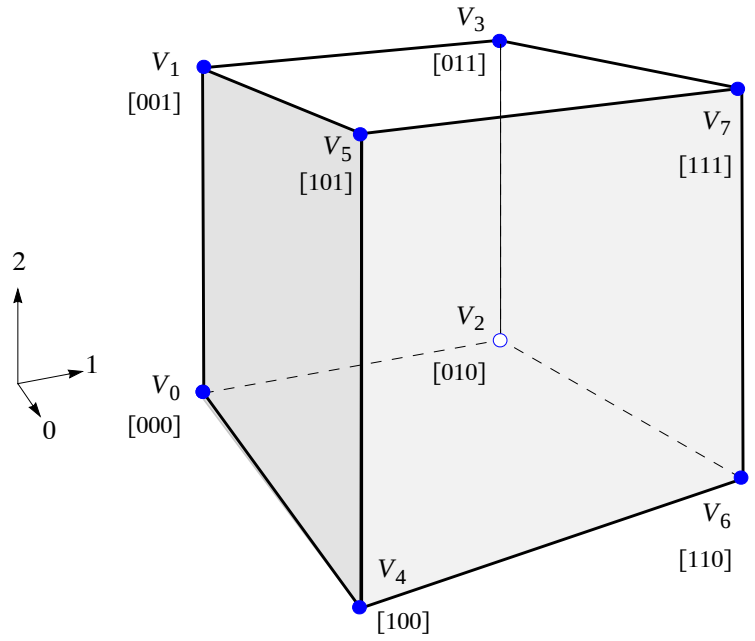


Figure 3-8: Vertex numbering with in a cell, numbers in square brackets [-] are the crystal directions of each vertex.

1. This is a choice of convenience. All three integer coordinates may, of course, be stored separately, which would give 2^{64} -1 = 1.84 x 10^{19} addressible integer locations in each coordinate direction.

2. Verified on MIPS R4000, HP9000, RISC R6000, SPARC and Pentium CPUs.

3. Such systems are quite general and can be used to describe cubic, orthorhombic, tetrahedral, or hexagonal cells. See [86].

$$
\begin{aligned}
\bar{V}_1 &= \bar{V}_0 + (\quad 0, \quad 0, 2^{Rmax_2 - R_2}) \\
\bar{V}_2 &= \bar{V}_0 + (\quad 0, 2^{Rmax_1 - R_1}, \quad 0) \\
\bar{V}_3 &= \bar{V}_0 + (\quad 0, 2^{Rmax_1 - R_1}, 2^{Rmax_2 - R_2}) \\
\bar{V}_4 &= \bar{V}_0 + (2^{Rmax_0 - R_0}, \quad 0, \quad 0) \\
\bar{V}_5 &= \bar{V}_0 + (2^{Rmax_0 - R_0}, \quad 0, 2^{Rmax_2 - R_2}) \\
\bar{V}_6 &= \bar{V}_0 + (2^{Rmax_0 - R_0}, 2^{Rmax_1 - R_1}, \quad 0) \\
\bar{V}_7 &= \bar{V}_0 + (2^{Rmax_0 - R_0}, 2^{Rmax_1 - R_1}, 2^{Rmax_2 - R_2})
\end{aligned}
\tag{3.8}
$$

Since the powers of two in this expression are simply a left shift of the bitwise representation of the integer subtraction $Rmax_j - R_j$, vertices V_1 through V_7 can be computed from V_0 and R_j at very low cost. In addition, the total number of refinements in each direction will be a (relatively) small integer, thus its possible to pack all three components of $\bar{R}$ into a single 32-bit word.

3.3 Boundary Conditions and Cut-Cell Intersection

3.3.1 Cut-Cell Boundary Fidelity

At wall boundaries, Cartesian cells are cut arbitrarily by the body geometry. This intersection may be described to the flow solver with various degrees of fidelity. Reference [3] categorized this description into three levels of approximation. Boundary conditions fitting this classification have been investigated by a variety of authors[3,18,40,67,70].

Level 1: The volume of each cut-cell which is inside the flow is computed by subtracting out the volume of the cell which protrudes into the wall. In the sketch in Figure 3-9.b this resulting cut-cell volume is

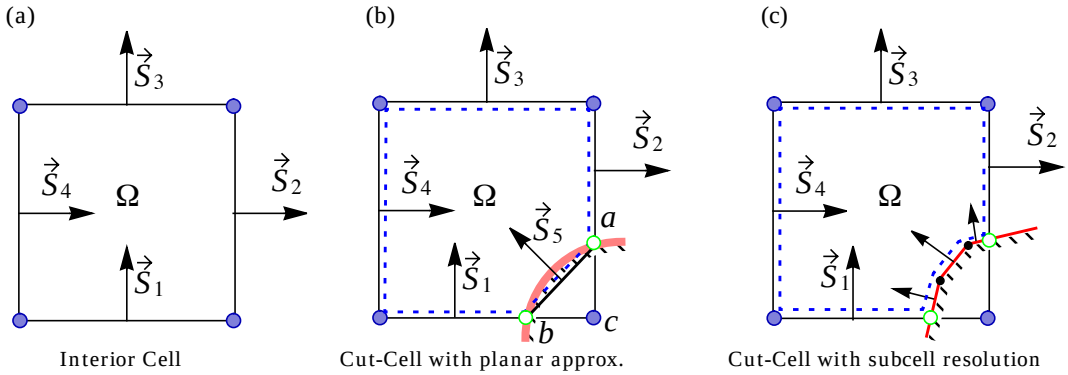


Figure 3-9: A general interior cell and two different models for surface cut-cells.

$V = V_\Omega - V_{\Delta abc}$. A single additional surface vector is kept to model the boundary which may be computed using the divergence theorem.

Level 2: In addition to (1), the intersection of the body surface with the cell is computed and stored. In three dimensions, this implies this information requires that the one store the intersections of the Cartesian edges with the body surface. The state vector for the flow computation is stored at the centroid of the flow-through region of the cut cell (the dotted region in Figure 3-9.b). Centroids of the cut-faces of the Cartesian cell may also be computed. The surface normal of the body is still inferred from a planar approximation.

Level 3: In addition to (2), sub-cell information about the variation of the surface within the cut cell is computed. The integration of the cut-cell follows a path conforming to the actual boundary (see Figure 3-9.c).

Numerical boundary conditions using all three levels of modeling have been formulated in the literature (see [3, 18, 32, 40, 70] among others). The investigations in [3] indicated that boundary conditions based on level 2 or 3 modeling may reduce the truncation error at the wall by as much as an order of magnitude over level 1 implementations, and reference [41] has demonstrated fully second-order boundary conditions for planar walls based on the information in level 2 modeling.

3.3.2 Cut-Cell/Surface Intersection

Implementation of these numerical boundary conditions requires the intersection of the body cut Cartesian cells with the surface triangulation. While the general edge-triangle intersection algorithm in section 2.2.2 offers one method of testing for such intersections, a more attractive alternative capitalizes on simplifications stemming from the fact that the target regions are Cartesian cells.

In three dimensions, the surface triangulation will cut arbitrarily through the body intersecting Cartesian cells. These intersections can be quite complex. We can begin to understand the details of such an intersection by considering the generic cut-cell illustrated in Figure 3-10. The abstraction shown in the sketch presents a single cut-cell, *c*, which is linked to a set $\{T_c\}$ of four triangles (T_0-T_3) which comprise the small swatch of the configuration's surface triangulation intersected by the cell. Since both the Cartesian cell and the triangles are convex, the intersection of each triangle with the cell produces a convex polygon referred to as a *triangle-polygon*, *tp*. Edges of the triangle-polygons are formed by the clipped edges of the triangles themselves, and the *face-segments, fs,* which result from the intersection of the triangles with the

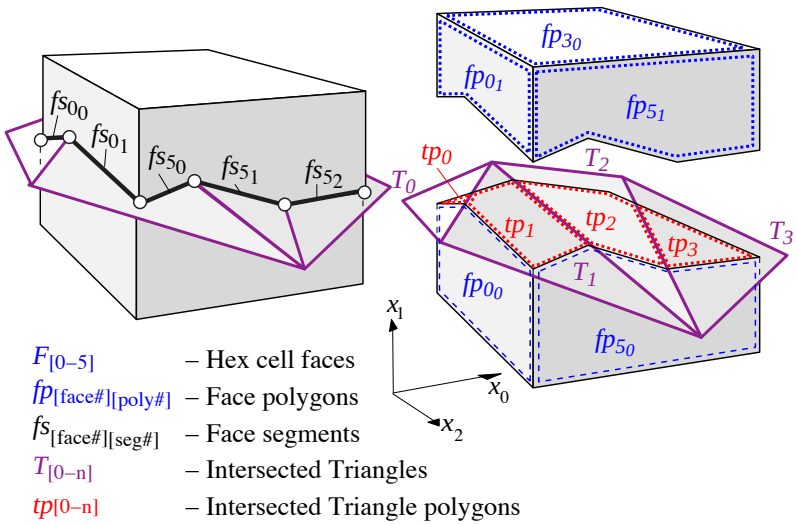


Figure 3-10: Anatomy of an abstract cut-cell.

faces of the Cartesian cell. The faces of the Cartesian cell may also be intersected by the surface triangles. Such intersections produce *face-polygons*, *fp*, which consist of edges from the cut-cell and face segments from the triangle-face intersection. Note that the face-polygons are not necessarily convex. This is demonstrated by the non-convex face-polygons $fp_{0,1}$, $fp_{5,0}$, and $fp_{5,1}$ in Figure 3-10. Supporting the entire hierarchy of boundary modeling discussed in the previous section requires us to fully specify each of these geometric entities for every cut Cartesian cell.

Obviously, these intersections may become very complex. Its easy to envision the pathological case where an entire configuration intersects only one or two Cartesian cells, creating tens of thousands of triangle polygons. Thus, an efficient implementation is of paramount importance. The algorithms for efficiently constructing this geometry rely on techniques from the literature on computer graphics and are highly specialized for use with coordinate aligned regions[30,87]. In principle, similar methods could be adopted for non-Cartesian hexahedra, or even other cell types, however, speed and simplicity would be compromised. Since rapid cut-cell intersection is an important part of Cartesian mesh generation, we present a few central operations in detail.

Rapid Intersection with Coordinate Aligned Regions

Figure 3-11 shows a two dimensional Cartesian cell *c* which covers the region $[\bar{c}, \bar{d}]$. The points (*p, q,...,v*) are assumed to be vertices of *c*'s candidate triangle list T_c. Each vertex is assigned an "outcode" associated with its location with respect to cell *c*. This

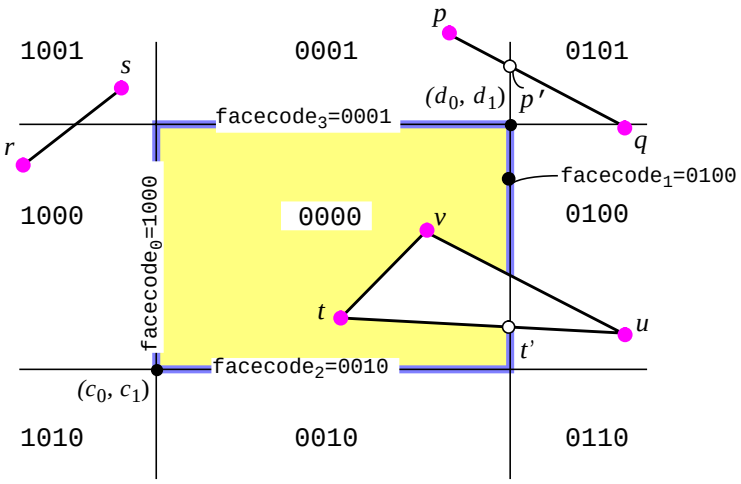


Figure 3-11: outcode and facecode setup of a coordinate aligned region $[\bar{c}, \bar{d}]$ in two dimensions.

code is really an array of flags which has a "low" and a "high" bit for each coordinate direction, $[\mathrm{lo}_0, \mathrm{hi}_0, \ldots, \mathrm{lo}_{d-1}, \mathrm{hi}_{d-1}]$. Since the region is coordinate aligned, a single inequality must be evaluated to set each bit in the outcode of the vertices. Points inside the region, $[\bar{c}, \bar{d}]$, have no bits set in their outcode.

Using the operators & and | to denote bitwise applications of the "and" and "or" boolean primitives, candidate edges (like $\overline{rs}$) can be trivially rejected if

$$\mathtt{outcode}_r \,\&\, \mathtt{outcode}_s \neq 0 \tag{3.9}$$

which reflects the fact that the outcodes of both r and s will have their low x bit se, thus neither point may be inside the region. Similarly, since ($\mathtt{outcode}_t$ | $\mathtt{outcode}_v$) = 0, the segment $\overline{tv}$ must be completely contained by the region $[\bar{c}, \bar{d}]$ in the figure.

If all the edges of a triangle, like Δ_{tuv}, cannot be trivially rejected, then there is a possibility that it intersects the 0000 region. Such a polygon can be tested against the face-planes of the region by constructing a logical bounding box (using a bitwise "or") and testing against each facecode of the region. In Fig. 3-11 testing

$$\mathtt{facecode}_j \,\&\, (\mathtt{outcode}_t \mid \mathtt{outcode}_u \mid \mathtt{outcode}_v) \quad \forall\, j \in \{0, 1, 2, \ldots, 2d\text{-}1\} \tag{3.10}$$

results in a non-zero only for the 0100 face. In eq. 3.10, the logical bounding box of Δ_{tuv} is constructed by taking the bitwise "or" of the outcodes of its vertices.

When a constructed intersection point, such as p' or t', is computed (with the method in §2.2.3), it can be classified and tested for containment on the boundary of $[\bar{c}, \bar{d}]$ by examination of its outcode. However, since these points lie degenerately on the `01XX` boundary, the contents of this bit may not be trustworthy. For this reason, we mask out the questionable bit before examining the contents of these outcodes. Applying "not" in a bitwise manner yields:

$$(\texttt{outcode}_{p'} \,\&\, (\neg\, \texttt{facecode}_1)) = 0 \qquad \text{while} \qquad (\texttt{outcode}_{t'} \,\&\, (\neg\, \texttt{facecode}_1)) \neq 0 \tag{3.11}$$

which indicates that t' is on the face, while p' is not.

There are clearly many alternative approaches for implementing the types of simple queries that this section describes. However, an efficient implementation of these operations is central to the success of a Cartesian mesh code. The bitwise operations and comparisons detailed in the proceeding paragraphs generally execute in a single machine instruction making this a particularly attractive approach.

Polygon Clipping

With the fast spatial comparison operators in the previous section outlined, we are ready to construct the triangle-polygons and face-segments which describe the surface within the Cartesian cell. The triangle-polygons (tp_0-tp_4) in Figure 3-10 result from the intersection of each triangle in $\{T_c\}$ with the Cartesian cell itself. Notice, however, that these triangle-polygons are not the formal intersection of the boundary of the Cartesian cell with the triangle, this intersection results in only the face-segments. The triangle-polygons are the regions of the triangles which lie *within* the cut-cells. Thus, extraction of the triangle-polygons is properly thought of as a *clipping* operation.

In the field of computer graphics, the term "clipping" refers to an intersection where one object acts as a "window" and we compute the parts of a second object visible through this window[39]. The most common type of clipping is when one object is a rectangle or a cube and various algorithms have been proposed for this case[53,61]. In this section we apply an algorithm due to Sutherland and Hodgman for clipping against any convex window[84]. While slightly more general than is absolutely necessary, this algorithm has the attractive property that the output polygon is kept as an ordered list of vertices which neatly maps into the winged-edge data structure discussed earlier (see §1.4.3).

The asymptotic complexity of this clipping algorithm is $O(pq)$, where p is the degree of the clip window and q is the degree of the clipped object. While this time bound is formally quadratic, p for a Cartesian cell is only 6, and the fast intersection checks of the previous section promote very effective filtering of trivial cases.

The Sutherland-Hodgman algorithm adopts a divide-and-conquer strategy which views the entire clipping operation as a sequence of identical, simpler problems. In this case the process of clipping one polygon against another is transformed into a sequence of clips against an infinite edge. Figure 3-12 illustrates the process for an arbitrary polygon clipped against a rectangular window. The input polygon is clipped against infinite edges constructed by extending the boundaries of the clip window.

Essentially the algorithm is implemented as two nested loops. The outer loop sweeps over the clip-border (cell faces in 3-D), while the inner is over the edges of the polygon. In our application to the intersected triangles, the initial input polygon is the triangle T, and the clip-window is the cut Cartesian cell. Implementation of the algorithm requires testing of the input triangle's edges against the clip region, so its useful to combine this algorithm with the outcode flags discussed in the previous section.

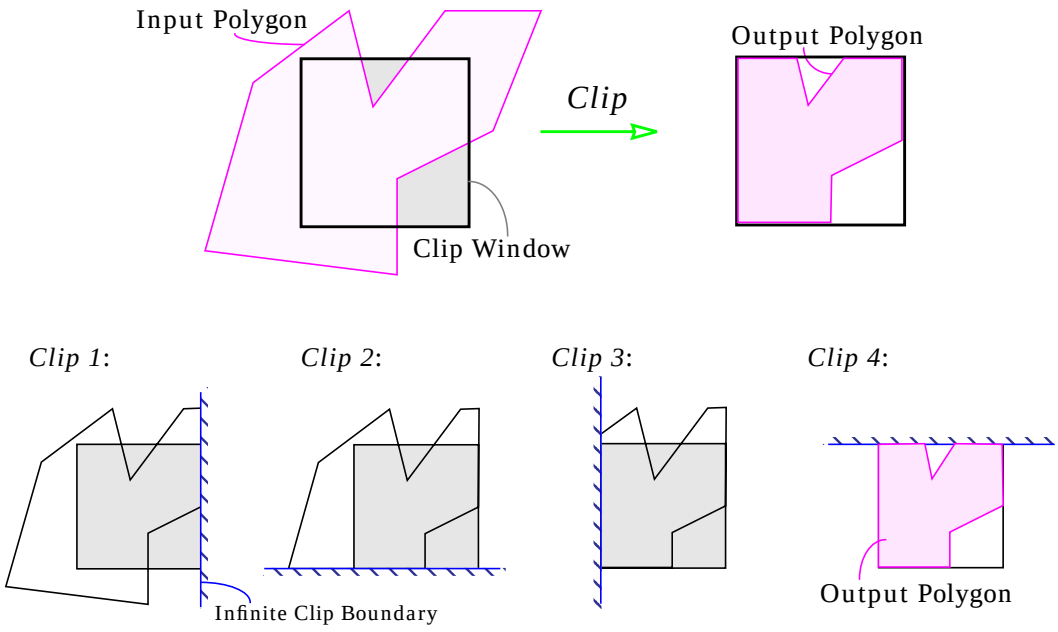


Figure 3-12: Illustration of divide-and-conquer strategy of Sutherland-Hodgman polygon clipping. The problem is recast as a series of simpler problems in which a polygon is clipped against a succession of infinite edges.

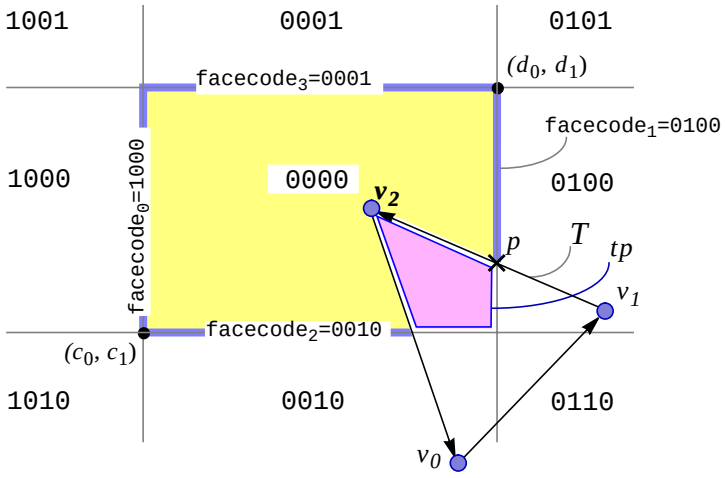


Figure 3-13: Setup for clipping a candidate triangle, *T*, against a coordinate aligned region and extracting the clipped triangle, *tp*.

Figure 3-13 illustrates the clipping problem for generating the triangle-polygons shown in the view of an abstract cut-cell in Figure 3-10 (page 89). In the sketch above, the triangle T is formed by the set of directed edges, $\overline{v_1 v_0}$, $\overline{v_2 v_1}$, and $\overline{v_0 v_2}$, and the clipped polygon, *tp*, is a quadrilateral.

As the edges of the input polygon are processed by each clip-boundary the output polygon is formed according to a set of four rules. For each directed edge in the input polygon we denote the vertex at the origin of the edge as "`origin`" and the vertex of the destination as "`destination`". "IN" implies that the test vertex is on the same side of the clip-boundary as the clip-window. We may test for this by examining the outcode of each vertex, and comparing to the facecode of the current-clip boundary. A test vertex is "IN" if its outcode does not have the bit associated with the facecode of the clip-boundary set, while "OUT" implies that this bit is set. Using the bitwise operators from the previous section:

$$\text{if (} \texttt{facecode}(\text{clip-boundary}) \;\&\; \texttt{outcode}(\text{vertex}) = 0 \text{) then IN} \qquad (3.12)$$
$$\text{if (} \texttt{facecode}(\text{clip-boundary}) \;\&\; \texttt{outcode}(\text{vertex}) \neq 0 \text{) then OUT}$$

With these definitions, the output polygon is constructed by traversing around the perimeter of the input polygon and applying the following rules to each edge:

SH.1. If ((`origin` is IN) and (`destination` is IN)) →Add `destination` to the output polygon.

SH.2. If ((`origin` is IN) and (`destination` is OUT)) →Add intersection of edge and clip-boundary to the output polygon.

SH.3. If ((`origin` is OUT) and (`destination` is OUT)) →Do nothing.

SH.4. If ((`origin` is OUT) and (`destination` is IN)) →Add both intersection and `destination` to output polygon.

Several aspects of these construction rules merit comment. Notice that both **SH.2** and **SH.4** consider cases where the edge of the input polygon crosses the clip-boundary. In both of these cases, we must add the point of intersection of the edge with the clip-boundary to the output polygon. This point may be easily constructed using the pierce-point constructor from §2.2.3, however, since our clip-boundary is coordinate aligned, we may dramatically simplify the eqs.2.6-2.8. For the example in Figure 3-13, the constructor for point p which is the intersection of edge $\overline{v_2 v_1}$ with the right side of the clip-boundary reduces to:

$$\vec{p} = \vec{v_1} + \alpha(\vec{v_2} - \vec{v_1}) \tag{3.13}$$

where α is simply the distance fraction in the horizontal coordinate of the clip boundary between vertices v_1 and v_2.

Returning to the cut-cell shown in Figure 3-10, we note that the face-segments are the edges of the triangle-polygons (just created) that result from a clip. The face-polygons are formed by simply connecting loops of cut-cell edges with these face-segments. Thus, all the necessary elements of the cut-cell have been constructed.

Since the Sutherland-Hodgman algorithm was originally developed for window clipping in computer graphics, both hardware and software versions of it are available on many platforms. Thus, on platforms with advanced graphics hardware, it is frequently possible to make direct calls to the hardware clipping routines to perform the polygon clipping discussed in the preceding paragraphs. Such hardware implementations typically execute tens to hundreds of times faster than software implementations. Similarly, many of the fast bitwise comparators in the previous section are often available as hardware routines.

Clipping Performance

Section 3.2.1 reasoned that by inheriting parent cell's triangle lists, the complexity of proximity searching for intersecting triangles would decrease as the cells refine and these lists become shorter. This same argument holds for the formation of the triangle-polygons within the cut-cells. As illustrated by Figure 3-3, as the cells refine, they are linked to fewer and fewer triangles, thus, there are fewer triangle-polygons to construct within each cut-cell. Table 3.1 chronicles the performance of the Sutherland-Hodgman routine for several example computations performed with the attack helicopter configuration shown in Figure 1-2 on page 7. The polygon clipping in this

example was performed using a software implementation of the Sutherland-Hodgman routine and the fast bitwise comparators.

An examination of the data in Table 3.1 reveals that, as predicted, the average time spent processing each cut-cell decreases as the mesh is refined. The average time to extract the triangle-polygons on the coarsest mesh (with 21578 cut-cells) was 7.6×10^{-5} sec/cut-cell. While cut-cells on the finest mesh were processed at a rate of 2.15×10^{-5} sec/cut-cell. On the finest mesh there were 387130 cut-cells which were linked to a total of approximately 930000 triangles, and clipping these triangles required a total of 8.33 sec. Thus the Sutherland-Hodgman algorithm had an average processing rate of just over 110000 triangles/second. A detailed profile of this algorithm revealed that the floating-point construction operation in eq. 3.13 accounted for fully one third of the processing time. This result indicates that the logic and bitwise operators in the clipping routines have relatively little overhead.

Table 3.1. Performance of clipping routine for creating triangle-polygons[a]

Number of Cut-cells	Avg. No. of Triangles/cut-cell	Total Time (sec)	Avg. Time/Cut-cell (sec)
21578	10.8	1.64	7.6×10^{-5}
43252	6.7	2.10	4.87×10^{-5}
99751	4.3	3.89	3.90×10^{-5}
387130	2.4	8.33	2.15×10^{-5}

a. Timings performed on 195 Mhz MIPS R10000 CPU.

Figure 3-14 shows an example of the intersection between the body-cut Cartesian cells and the surface triangulation of a High Wing Transport configuration. In this case approximately 500000 cells in the Cartesian mesh intersected the surface triangulation. The figure shows a view of the port side of the aircraft and two zoom-boxes with successive enlargements of the triangle-polygons resulting from the intersection. In this example, the triangle-polygons have (themselves) been triangulated before plotting. This example contained about 2.9M cells in the full Cartesian mesh.

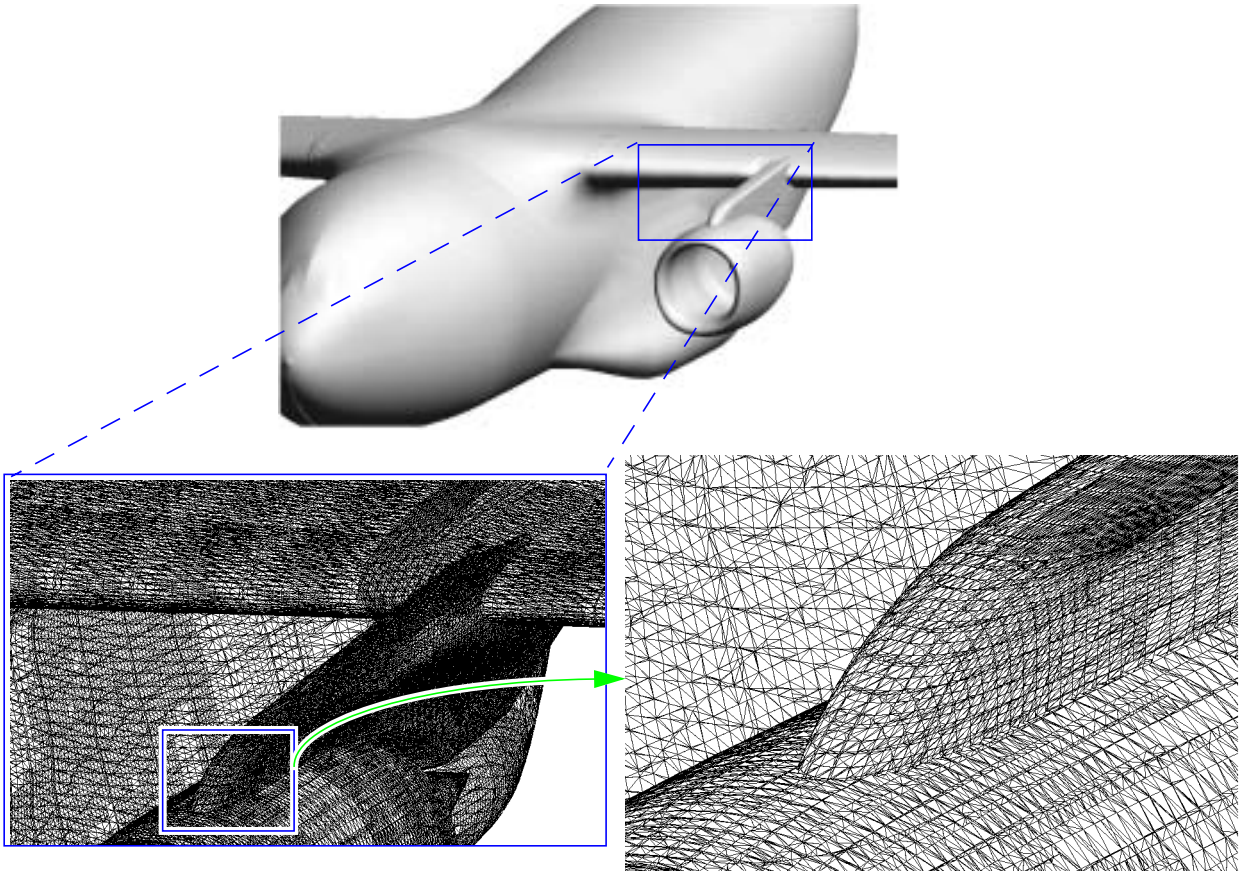


Figure 3-14: Triangle-polygons on surface of High Wing Transport configuration resulting from intersection of body-cut Cartesian cells with surface triangulation. The triangle-polygons shown have been (themselves) triangulated for plotting purposes. This example included approximately 500000 body-cut Cartesian cells.

3.4 Example Cartesian Meshes

The intersection algorithm described in section 2 and the mesh generator strategy covered in the preceding paragraphs have been exercised on a variety of example problems. All of the computations presented here were performed on a MIPS R10000 workstation with a 195Mhz CPU.

High Speed Civil Transport (HSCT)

Figure 3-15 depicts three views of a 4.72M cell mesh constructed around a proposed supersonic transport design. This geometry consists of 8 polyhedra, two of which have non-zero genus. These components include the fuselage, wing, engine pylons and nacelles. The original component triangulation was comprised of 81460 triangles before intersection and 77175 after the intersection algorithm re-triangulated the

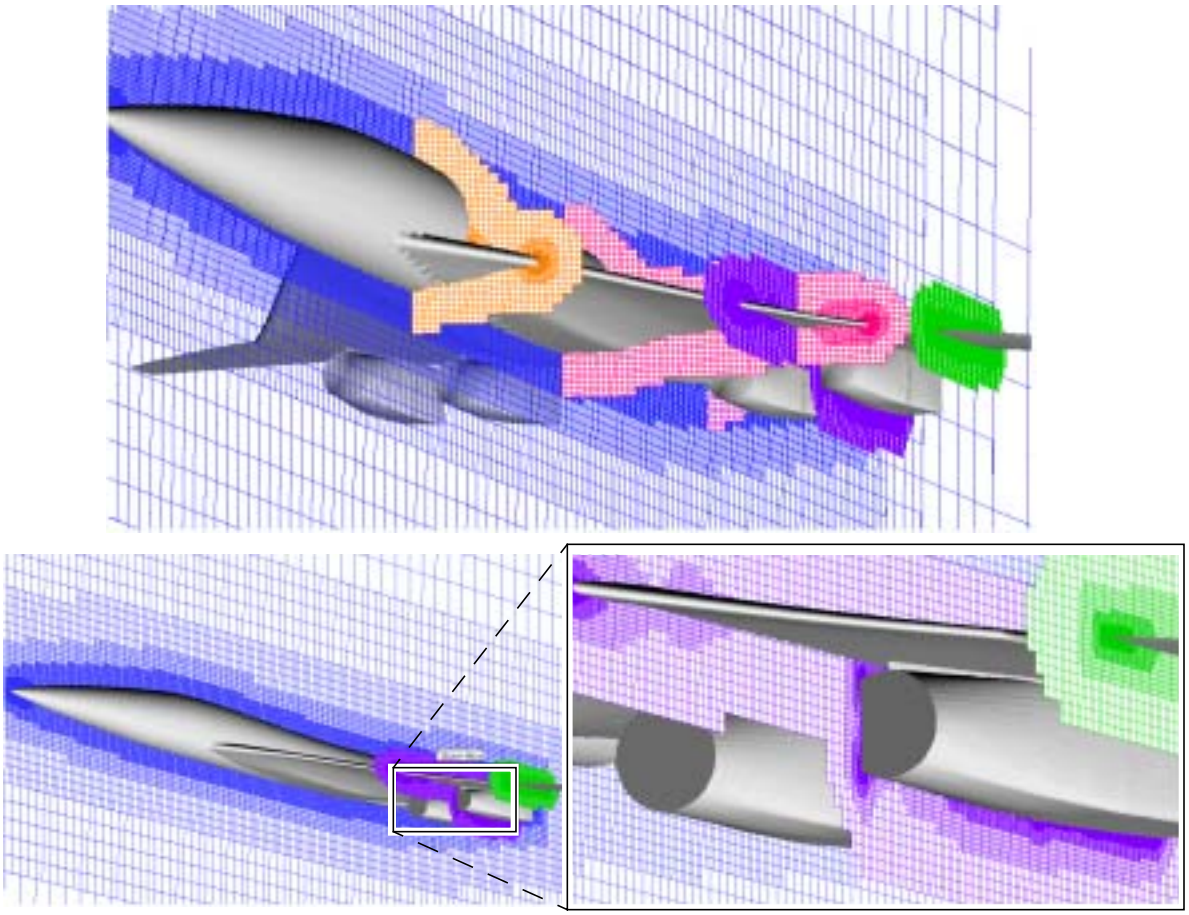


Figure 3-15: Upper: Cutting planes through 4.72M cell Cartesian mesh for a proposed HSCT geometry. Lower: Close-up of mesh near outboard nacelle.

intersections and extracted the wetted surface. The component intersection placed 1.2M calls placed to the determinant computation (eq. 2.3), 1037 of these invoked the exact arithmetic and of these, 870 were truly degenerate and required tie-breaking by virtual perturbation. The intersection required 15 seconds of workstation time.

The mesh shown contains 11 levels of cells where all divisions were isotropic. Mesh generation required 4 minutes and 20 seconds. The maximum memory required was 252Mb.

Multiple Aircraft Configuration

The final configuration begins with the attack helicopter example from Figure 1-2 on page 7, and then adds three twin-tailed fighter models to the geometry. The helicopter in this example is off-set from the axis of the lead fighter to emphasize the asymmetry of the mesh. Each fighter has flow-through inlets and is described by 13

component triangulations. The entire configuration contained 121 components described with 807000 triangles before the component intersection of section 2, and 683000 triangles after the internal geometry was trimmed out. A total of 5916 determinant evaluations (eq. 2.3) were identified as degenerate by the exact arithmetic routines and invoked the virtual perturbation routines described in section.

Figure 3-16 presents two views of the final mesh. The upper frame shows portions of 3 cutting planes through the geometry. The lower frame in this figure shows one cutting plane at the tail of the rear two aircraft, and another just under the helicopter geometry. The volume mesh includes 5.61M cells, of which 488000 intersect the geometry. This case required a maximum of 365Mb to compute, including storage of

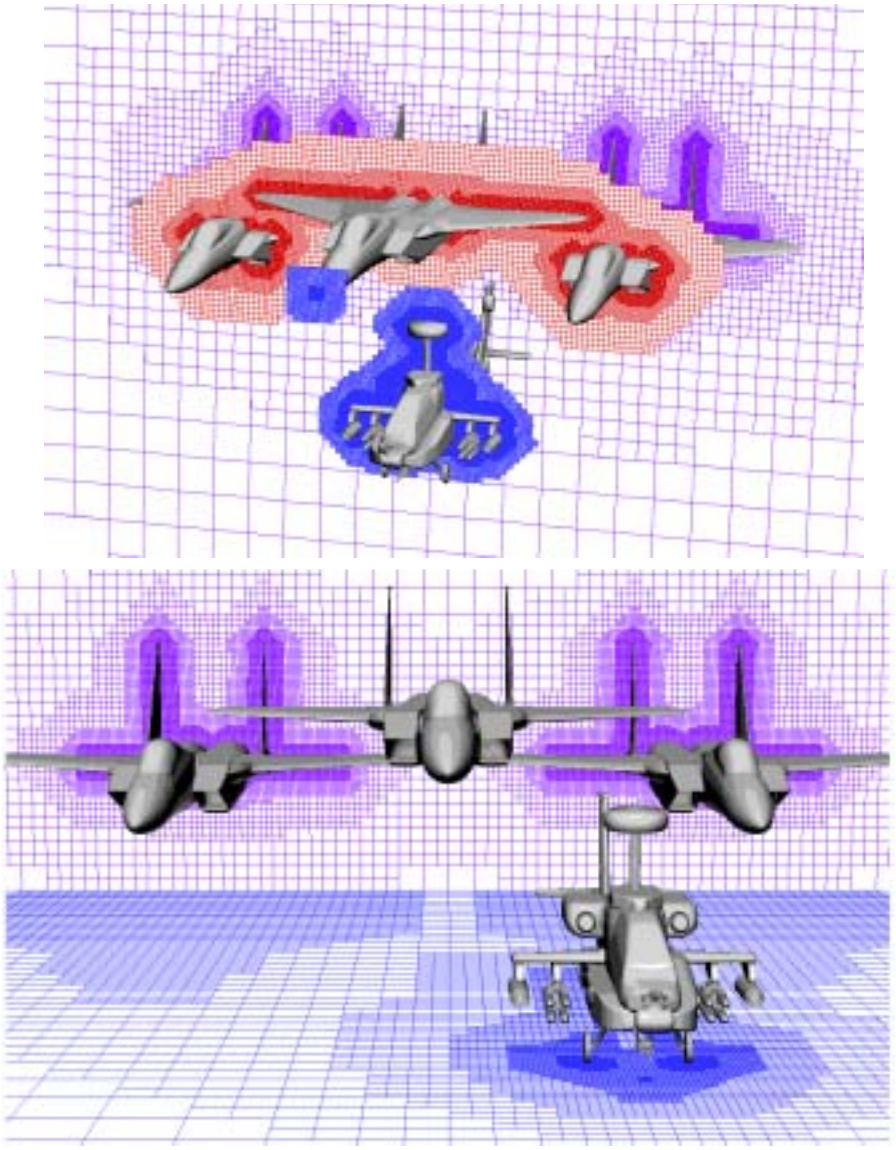


Figure 3-16: Cutting planes through mesh of multiple aircraft configuration with 5.61M cells and 683000 triangles in the triangulation of the wetted surface.

the surface triangulation and the volume mesh. The total mesh generation time was approximately 6 minutes and 30 seconds which includes approximately 15 seconds for the formation of the polygon clipping described in the previous section (§ 3.3.2).

3.5 Asymptotic Performance

Section 1.3.4 briefly touched on the topic of asymptotic complexity for unstructured Cartesian mesh methods. Two of the primary factors effecting mesh generation speed are the number of triangles in the surface description and the percentage of the mesh cells which actually intersect this boundary. The examples in the previous section have been chosen to demonstrate mesh generation speed on realistically complex geometries.

Figure 1-4 on page 10 presented a scatter plot of mesh size vs. CPU time in order to assess the asymptotic behavior of the algorithm. For that investigation, the mesh generator was run on a teardrop geometry described by 7520 triangles. To prevent variation in the percentage of cut cells which are divided at successive refinements, the angle thresholds triggering mesh refinement were set to zero. This choice forced all cut cells to be tagged for refinement at every level.

Results from a series of 11 meshes are displayed in that figure, these contained between 7.5×10^3 and 1.7×10^6 cells in the final grids. The initial meshes used consisted of 6x6x6, 5x5x6, and 5x5x5 cells and were subjected to 3-9 levels of refinement. As shown by the plot in Figure 1-4 this investigation demonstrated linear asymptotic performance and produced cells ar a rate of approximately 24950 cells-per-second (1.50×10^6 cells-per-minute). See the discussion in section 1.3.4 for further details.

3.6 Future Work

One aspect which has not been completely addressed is the degree to which anisotropic cell division can be used to improve the efficiency of adaptive Cartesian simulations on realistic geometries. Since the current method has the ability to refine cells directionally, this topic will be addressed in future work.

The other outstanding issue surrounding adaptive Cartesian mesh methods concerns, of course, viscous approaches. While some progress has been made in hybrid Cartesian-prismatic meshes, results have not been completely satisfying, and research continues on this topic.

Acknowledgments

Much of the work described in these notes was done in collaboration with my co-investigators Marsha Berger (Courant Institute) and John Melton (NASA Ames). I thank Marsha for several of the analyses included in these notes as well as many long hours of writing (and debugging) C code. John gets credit not only for initially introducing me to the possibilities of Cartesian mesh methods, but also for either initiating or contributing to many of the ideas that have been presented in these notes. John's research and thesis work has provided motivation for many of the algorithms discussed in the preceding pages.

I would also like to thank Ken Powell, Eric Charlton, and Sami Bayyuk for many thoughtful discussions and permission to reprint some of their results in these notes. Ken's contributions - both personally and through his students - have had a major effect on our Cartesian grid work at Ames.

In addition, I am grateful to Diane Poirier, and Datta Gaitonde for their long hours spent proofreading and helping to clarify some of the discussions and arguments presented in these pages. Finally, I thank Jonathan Shewchuk both for discussions on floating-point error estimation and the use of his adaptive-precision floating-point library.

References

[1] Aftosmis, M.J., "Emerging CFD Technologies and Aerospace Vehicle Design," *NASA Wkshp. on Surf. Mod., Grid Gen., and Related Issues in CFD,* NASA Lewis Rsch Cntr., May 9-11, 1995.

[2] Aftosmis, M.J., Berger, M.J., Melton, J.E., "Robust and efficient Cartesian mesh generation for component-based geometry," *AIAA Paper 97-0196,* Jan. 1997.

[3] Aftosmis, M.J., Melton, J.E., and Berger, M.J., "Adaptation and Surface Modeling for Cartesian Mesh Methods," AIAA Paper 95-1725-CP, Jun., 1995.

[4] AGARD Fluid Dynamics Panel, "Test cases for inviscid flow field methods," *AGARD Advisory Report AR-211*. May 1985.

[5] Almgren, A.S., Bell, J., Colella, P., and Howell, L., "An adaptive projection method for the incompressible Navier-Stokes equations," *Proc. IMACS 14th World Conference,* Atlanta, GA, July 1994.

[6] Almgren, R., and Almgren, A.S., "Phase field instabilities and adaptive mesh refinement," *Modern Methods for Modeling Microstructure in Materials.* TMS-SIAM, Oct 1995.

[7] Aurenhammer, F., "Voronoi diagrams: A survey of a fundamental data structure," *ACM Comput. Surveys* **23**:345-405, 1991.

[8] Bailey, D.H., *A Portable High Performance Multiprecision Package.* NASA TR-RNR-90-022

[9] Barth, T.J., *Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations*, von Karman Institute for Fluid Dynamics, Lecture Series 1994-05, Rhode-Saint-Genèse, Belgium, Mar. 21-25, 1994.

[10] Barth, T.J., and Jespersen, D.C., "The design and analysis of upwind schemes on unstructured meshes," *AIAA Paper 89-0366,* Jan. 1989.

[11] Baumgart, B. G., "A polyhedron representation for computer vision." in *Proc. AFIPS Natl. Comp. Conf.,* **44**:589-596, 1975.

[12] Bayyuk, S., *Euler Flows with Arbitrary Geometries and Moving Boundaries.* Ph.D thesis, Dept. of Aero. and Mech. Eng., Univ. of Mich., 1996.

[13] Berger M.J., Aftosmis, M.J., and Melton, J.E., "Accuracy, adaptive methods and complex geometry," *Proc. 1st AFOSR Conf. on Dynam. Mot. in CFD.* Rutgers, NJ 1996.

[14] Berger M.J., and Colella, P., "Local adaptive mesh refinement for shock hydrodynamics." *Jol. of Comp. Physics,* **82**:64-84, 1989

[15] Berger, M.J., and Jameson, A., "An adaptive multigrid method for the Euler equations," *Lecture Notes in Physics,* 218, 1984.

[16] Berger, M.J., and LeVeque, R., "Stable Boundary Conditions for Cartesian Grid Calculations", *ICASE Report No. 90-37*, 1990.

[17] Berger, M.J., and LeVeque, R., "Cartesian Meshes and Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations", *Proc. 3rd Intl. Conf. Hyp. Problems,* Uppsala, Sweden, 1990.

[18] Berger, M.J., and Melton, J.E., "An Accuracy Test of a Cartesian Grid Method for Steady flow in Complex Geometries," Proc. *8thrIntl. Conf. Hyp. Problems,* Uppsala, Stonybrook, NY, Jun., 1995. also RIACS Report 95-02.

[19] Berger, M.J., and Oliger, J., "Adaptive mesh refinement for hyperbolic partial differential equations." *Jol. of Comp. Physics.* **53**:482-512, 1984.

[20] Bonet, J., and Peraire, J., "An alternating digital tree (ADT) Algorithm for Geometric Searching and Intersection Problems." *Int. J. Num. Meth. Eng*, **31**:1-17, 1991.

[21] Boris, J., "A vectorised algorithm for determining the nearest neighbours." *Jol. Comp. Phy.*, **66**:1-20, 1986.

[22] Bowyer, A., "Computing Dirichlet tessellations," *The Computer Jol.* **24**(2):162-166, 1981

[23] Buell, D.A., and Ward, R.L., "A multiprecise integer arithmetic package," *Jol. of Supercomputing*, **3**:89-107, 1989.

[24] Chan, W. M. and Meakin, R. L., "Advances towards automatic surface domain decomposition and grid generation for overset grids," *AIAA Paper 97-1979*, in Proceed. of the AIAA 13th Comp.Fluid Dyn. Conf., Snowmass, Colorado, Jun. 1997.

[25] Charlton. E.F., and Powell, K.G., "An octree solution to conservation-laws over arbitrary regions (OSCAR)." *AIAA Paper 97-0198*, Jan. 1997.

[26] Chazelle, B., et al., *Application Challenges to Computational Geometry: CG Impact Task Force Report*. TR-521-96. Princeton Univ., Apr. 1996.

[27] Chew, L.P., "Constrained Delaunay triangulations," Algorithmica, **4**:97-108, 1989.

[28] Chvátal, V., *Linear Programming*. Freeman, San Francisco, Ca., 1983.

[29] Clarke, D., Salas, M., and Hassan,H., "Euler Calculations for Multi-Element Airfoils using Cartesian Grids," *AIAA Jol*, **24**, 1986.

[30] Cohen, E., "Some Mathematical Tools for a Modeler's Workbench," IEEE Comp. Graph. and App., **3**(7), Oct. 1983.

[31] Coirier, W.J., "An Adaptively-Refined, Cartesian, Cell-Based Scheme for the Euler Equations," *NASA TM-106754*, Oct., 1994. also Ph.D. Thesis, Univ. of Mich., Dept. of Aero. and Astro. Engr., 1994.

[32] Coirier, W. J., and Powell, K. G., "An Accuracy Assessment of Cartesian-Mesh Approaches for the Euler Equations", *AIAA Paper 93-3335-CP*, July, 1993.

[33] Delaunay, B., "Sur la Sphére Vide," Izvestia Akademii Nauk SSSR, **7**(6):793-800, Oct. 1934.

[34] De Zeeuw, D., and Powell, K., "An Adaptively-Refined Cartesian Mesh Solver for the Euler Equations," *AIAA Paper 91-1542*, 1991.

[35] DT_NURBS Spline Geometry Subprogram Library Theory Document, version 3.3. USN Surface Warfare Center/Carderock Div. David Taylor Model Basin, Bethesda MD. CARDEROCKDIV-94/000, Dec. 1996. see also `http://dtnet33-199.dt.navy.mil/dtnurbs/doc.htm`

[36] Edelsbrunner H., and Mücke E.P., "Simulation of Simplicity: A Technique to cope with degenerate cases in geometric algorithms." *ACM Transactions on Graphics*, **9**(1):66-104, Jan. 1990.

[37] Finkel, R.A., and Bentley, J.L., "Quad trees: a data structure for retrieval on composite keys," *Acta Informatica*, **4**(1):1-9, 1974.

[38] Floyd, R.W., and Knuth, D.E., "Addition machines," *SIAM Jol. of Computing*, **19**(2):329-340, 1990.

[39] Foley, J., van Dam, A., Feiner, S., Hughes, J., *Computer Graphics: Principles and Practice*, ISBN 0-201-84840-6, Addison-Wesley, Reading, MA, 1995.

[40] Forrer, H., *Boundary Treatment for a Cartesian Grid Method*, Seminar für Angewandte Mathmatic, ETH Zürich, ETH Research Report No. 96-04, 1996. see `http://www.sam.math.ethz.ch/Reports/1996-04.html`

[41] Forrer, H., *Second Order Accurate Boundary Treatment for Cartesian Grid Methods*, Seminar für Angewandte Mathmatic, ETH Zürich, ETH Research Report No. 96-13, 1996. see `http://www.sam.math.ethz.ch/Reports/1996-13.html`

[42] Gaffney, R., Hassan, H., and Salas, M., "Euler Calculations for Wings Using Cartesian Grids," *AIAA Paper 87-0356*, Jan.,1987.

[43] Gooch, C.F., "Solution of the Navier-Stokes Equations on Locally-Refined Cartesian Meshes," Ph.D. Dissertation, Dept. of Aero. Astro. Stanford Univ., Dec., 1993.

[44] Green, P.J., and Sibson, R., "Computing the Dirichlet Tessellation in the Plane." *The Computer Journal*, **2**(21):168-173, 1977.

[45] Grossman, B., and Whitaker, D., "Supersonic Flow Computations using a Rectangular-Coordinate Finite-Volume Method," *AIAA Paper 86-0442*, Jan., 1986.

[46] Guibas, L.J., Knuth, D.E., and Sharir, M., "Randomized incremental construction of Delaunay and Voronoi Diagrams", *Algorithmica* **7**(4):381-413, 1992.

[47] Karman, S.L.Jr., "Splitflow: A 3D unstructured Cartesian/prismatic grid CFD code for complex geometries," *AIAA 95-0343*, Jan., 1995.

[48] Knuth, D., *The Art of Computer Programming: Seminumerical Algorithms, Vol.2.*, Addison Wesley, 1973.

[49] Knuth, D., *The Art of Computer Programming: Sorting and Searching, Vol.3.*, Addison Wesley, 1973.

[50] Knuth, D.E., *Axioms and Hulls*. Lecture Notes in Comp. Sci. #606., Springer-Verlag, Heidelberg, 1992.

[51] Lednicer, D., Tidd, D., and Birch, N., "Analysis of a Close Coupled Nacelle Installation using a Panel Method (VSAERO) and a Multigrid Euler Method (MGAERO)," *ICAS-94-2.2.1*, 1994.

[52] Levy, D., Wariner, D., and Nelson, E., "Validation of Computational Euler Solutions for a High Speed Business Jet", *AIAA 94-1843*, Jun., 1994.

[53] Liang, Y., and Barsky, B.A., "An analysis and algorithm for polygon clipping," *Comm. of the ACM*, **26**(3):868-877, 1983.

[54] Löhner, R., "Regridding surface triangulations," Jol. Comp. Phy. **126**:1-10, 1996.

[55] Mavriplis, D.J., *Unstructured Mesh Generation and Adaptivity*, von Karman Institute for Fluid Dynamics, Lecture Series 1994-05, Rhode-Saint-Genèse, Belgium, Mar. 13-17, 1995.

[56] Melton, J.E., *Automated Three-Dimensional Cartesian Grid Generation and Euler Flow Solutions for Arbitrary Geometries*, Ph.D. thesis, Univ. CA. Davis CA, 1996.

[57] Melton, J.E., Berger, M.J., Aftosmis, M.J., and Wong, M.D., "3D Applications of a Cartesian Grid Euler Method," *AIAA Paper 95-0853*, Jan., 1995.

[58] Melton, J.E., Enomoto, F.Y., and Berger, M.J., "3D Automatic Cartesian Grid Generation for Euler Flows," *AIAA Paper -93-3386-CP*, Jul., 1993.

[59] Minion, M., *Two Methods for the Study of Vortex Patch Evolution on Locally Refined Grids.*, Ph.D. thesis, Univ. CA Berkeley, May 1994.

[60] Mücke, E.P., Saias, I., and Zhu, B., "Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations," *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, ACM, May 1996.

[61] Newman, W.M., Sproull, R.F., *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, NY, 1979.

[62] Ni, R.H., "A multiple grid scheme for solving the Euler equations," *AIAA Jol.*, **20**(11):1565-1571, Nov. 1982.

[63] Noh, W., Gee, M., and Kramer, G., Technical Report UCID-18515, Lawerence Livermore National Laboratory, 1979.

[64] O'Rourke, J., *Computational Geometry in C*, Cambridge Univ. Press, NY, 1993.

[65] *PARASOLIDS*. see `http://www.edsug.com/`

[66] Pember R., *et al.* "The modeling of a laboratory natural gas-fired furnace with a higher-order projection method for unsteady combustion," TR UCRL-JC-123244, Lawerence Livermore National Lab., Feb. 1996.

[67] Pember, R.B., Bell, J.B., Colella, P., Crutchfield, W.Y., and Welcome, M.L., "An adaptive Cartesian grid method for unsteady compressible flow in irregular regions," *Jol. of Comp. Phy.*, **120**:278-304, 1995.

[68] Pember, R.B., Greenough, J., and Colella, P., "An adaptive, higher-order Godunov method for gas dynamics in three-dimensional orthogonal curvilinear coordinates," TR UCRL-JC-123351, Lawerence Livermore National Lab., Feb. 1996.

[69] Peraire, J., and Morgan, K., "Viscous unstructured mesh generation using directional refinement," *Proc. of the 5th Num. Grid Gen in CFD and Related Fields Conf.*MS, Apr. 1996.

[70] Powell, K., *Solution of the Euler and Magnetohydrodynamic Equations on Solution-Adaptive Cartesian Grids*, von Karman Institute for Fluid Dynamics, Lecture Series 1994-05, Rhode-Saint-Genèse, Belgium, Mar. 1996.

[71] Preparata, F.P., and Shamos, M.I., *Computational Geometry: An Introduction*, Springer-Verlag, 1985

[72] Priest, D.M., "Algorithms for arbitrary precision floating point arithmetic," Tenth Symposium on Computer Arithmetic, pp. 132-143, IEEE Comp. Soc.Press, 1991.

[73] Purvis, J., and Burkhalter, J., "Prediction of Critical Mach Number for Store Configurations", *AIAA Jol.* **17(11)**, 1979.

[74] Quirk, J., "An alternative to unstructured grids for computing gas dynamic flows around arbitrarily complex two dimensional bodies," *ICASE Report 92-7*, 1992.

[75] Reed, K., "The Initial Graphics Exchange Specification (IGES) Version 5.1", Sept. 1991.

[76] Roma, A., *A Multilevel Self Adaptive Version of the Immersed Boundary Method*, Ph.D. thesis, NY Univ., Jan. 1996.

[77] Samet, H., *The Design and Analysis of Spatial Data Structures. Addison-Wesley Series on Computer Science and Information Processing.* Addison-Wesley Publishing Company, 1990.

[78] Schaudt, B., and Drysdale, R.L., "Multiplicatively weighted crystal growth Voronoi diagrams," in *Proc. 7th Ann. Symp. Comp. Geom.* ACM, pp.214-223, 1991.

[79] Sedgewick, R., *Algorithms*, Addison Wesley, Reading. 1988.

[80] Shewchuk, J.R., "Robust adaptive floating-point geometric predicates," Proceedings of the Twelfth Annual Symposium on Computational Geometry, pp.141-150, ACM, May 1996.

[81] Shewchuk, J.R., "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates." CMU-CS-96-140, School of Computer Science, Carnegie Mellon Univ., 1996. currently also available at:
`http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-predicates.ps`

[82] Sloan S.W., "A fast algorithm for generating constrained Delaunay triangulations," *Computers and Structures*, Pergammon Press Ltd., **47**(3):441-450, 1993.

[83] Sterbenz, P.H., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[84] Sutherland, I.E., and Hodgman, G.W., “Reentrant polygon clipping,” *Comm of the ACM*, **17**(1):32-42, 1974.

[85] Tidd, D. M., Strash, D. J., Epstein, B., Luntz, A., Nachson, A., and Rubin, T., “Application of an Efficient 3-D Multigrid Euler Method (MGAERO) to Complete Aircraft Configurations”, *AIAA Paper 91-3236*, Jun., 1991.

[86] Van Vlack, L.H., *Elements of Material Science and Engineering*, Addison-Wesley Inc., 1980.

[87] Voorhies, D., *Graphics Gems II: Triangle-Cube Intersections*. Academic Press, Inc. 1992.

[88] Voronoi, G., “Nouvelles applications des paramétres continus á la théorie des formes quadratiques,” *Jol. Reine Angew. Math*. **133**:97-178, 1907.

[89] Wang, Z.J., Przekwas, A., and Hufford, G., “Adaptive Cartesian/adaptive prism grid generation for complex geometry,” *AIAA Paper 97-0860*, Jan. 1997.

[90] Watson, D.F., “Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes,” *The Computer Jol.* **24**(2);167-171, 1981.

[91] Welterlen, T.J., and Karman, S.L.Jr., “Rapid Assessment of F-16 Store Trajectories Using Unstructured CFD,” *AIAA 95-0354*, Jan., 1995.

[92] Wendroff, B., and White, A., “Supraconvergent schemes for hyperbolic equations on irregular grids,” *Notes on Numerical Fluid Mech*. **24**, 1989.

[93] Woodward, P., “Piecewise parabolic methods for astrophysical fluid dynamics.” In K. -H., Winkler and Norman M. editors, *Astrophysical Radiation Hydrodynamics*, 1986.

[94] Yap, C-.K., “Geometric consistency theorem for a symbolic perturbation scheme,” *Jol. of Comp. and Sys. Sci*. **40**(1):2-18, 1990.

[95] Yap, C.-K., “Symbolic Treatment of geometric degeneracies,” *Jol. Symbolic Comput*, **10**:349-370, 1990.

[96] Yap, C., Dubé, T., “The exact computation paradigm,” *Computing in Euclidean Geometry*, (2nd Ed.), Eds, D.-Z. Du, and F.K. Hwang, World Scientific Press, pp. 452-492, 1995.